FIWARE-Stream-Oriented-GE

Release 6.8.0

Nov 09, 2018

Contents

The Stream Oriented Generic Enabler (GE) provides a framework devoted to simplify the development of complex interactive multimedia applications through a rich family of APIs and toolboxes. It provides a media server and a set of client APIs making simple the development of advanced video applications for WWW and smartphone platforms. The Stream Oriented GE features include group communications, transcoding, recording, mixing, broadcasting and routing of audiovisual flows. It also provides advanced media processing capabilities involving computer vision, video indexing, augmented reality and speech analysis.

The Stream Oriented GE modular architecture makes simple the integration of third party media processing algorithms (i.e. speech recognition, sentiment analysis, face recognition, etc.), which can be transparently used by application developers as the rest of built-in features.

The Stream Oriented GE's core element is a Media Server, responsible for media transmission, processing, loading and recording. It is implemented in low level technologies based on GStreamer to optimize the resource consumption. It provides the following features:

- Networked streaming protocols, including HTTP (working as client and server), RTP and WebRTC.
- Group communications (MCUs and SFUs functionality) supporting both media mixing and media routing/dispatching.
- Generic support for computational vision and augmented reality filters. - Media storage supporting writing operations for WebM and MP4 and playing in all formats supported by GStreamer.
- Automatic media transcodification between any of the codecs supported by GStreamer including VP8, H.264, H.263, AMR, OPUS, Speex, G.711, etc.

CHAPTER 1

Why Using Kurento in a “Smart Solution”?

The Stream-oriented GE provides a suitable structure to multimedia information, so it can be inserted into the context in an homogeneous way and can be consumed by client application front-ends or application backends just like any other context information.

Information can be extracted to convert media devices like cameras into IoT devices using the Kurento real-time media Stream processing GE. Context information can be generated as a result of the media streams analysis or the reception of context data to take decisions in the way the media is processed.

1.1 KURENTO QUICK START GUIDE

Welcome to the FIWARE Stream GE: Kurento! Here is what you need to do to start working with Kurento.

1. **Install KMS and “Built-in modules*”** The installation guide explains different ways in which Kurento can be installed and how to install any built-in modules you would need.

2. **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.

3. **Install and configure Orion** You want to make a Smart Solution, so you need to manage the context so you would want to use Orion Context Broker. Check the Orion Context Broker Installation & Administration Guide.

4. **Write an Application** Write an application that queries the Kurento API to make use of the capabilities offered by KMS. The easiest way of doing this is to build on one of the provided Kurento Clients. And integrate it with Orion Context Broker. In general, you can use *any programming language* to write your application, as long as it speaks the Kurento Protocol and it's able to use the REST API of Orion. Have a look at the features offered by Kurento, and follow any of the multiple tutorials that explain how to build basic applications.

5. **Ask for help** If you face any issue with Kurento itself or have difficulties configuring the plethora of mechanisms that form part of WebRTC, don't hesitate to ask for help to the Kurento community of users.

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 :-)

*** built-in modules** are extra modules developed by the Kurento team to enhance the basic capabilities of Kurento Media Server.

1.2 FIWARE Stream Oriented Generic Enabler - Installation and Administration Guide

This guide describes how to install the Stream-Oriented GE - Kurento. Kurento's core element is the **Kurento Media Server** (KMS), responsible for media transmission, processing, loading and recording. It is implemented in low level technologies based on GStreamer to optimize the resource consumption.

1.2.1 Introduction

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

For other OS and versions check *Running Kurento from a Docker container*

1.2.1.1 Requirements

To guarantee the right working of the enabler RAM memory and HDD size should be at least:

- 4 GB RAM

- 16 GB HDD (this figure is not taking into account that multimedia streams could be stored in the same machine. If so, HDD size must be increased accordingly).

1.2.2 Installation

1.2.2.1 KMS

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

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

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

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

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

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

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

3. Install KMS:

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

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

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

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

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

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

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

```
sudo netstat -tupan | grep kurento

> tcp6  0  0 :::8888  :::*  LISTEN  1270/kurento-media-server
```

1.2.2.2 Built-in modules

Built-in modules are extra modules developed by the Kurento team to enhance the basic capabilities of Kurento Media Server. So far, there are four built-in modules, that are installed as follows:

- **kms-pointerdetector**: Filter that detects pointers in video streams, based on color tracking.

```
sudo apt-get install kms-pointerdetector
```

- **kms-chroma**: Filter that takes a color range in the top layer and makes it transparent, revealing another image behind.

```
sudo apt-get install kms-chroma
```

- **kms-crowddetector**: Filter that detects people agglomeration in video streams.

```
sudo apt-get install kms-crowddetector
```

- **kms-platedetector**: Filter that detects vehicle plates in video streams.

```
sudo apt-get install kms-platedetector
```

1.2.3 Running Kurento from a Docker container

Starting a Kurento media server instance is easy. Kurento media server exposes port 8888 for client access. So, assuming you want to map port 8888 in the instance to local port 8888, you can start kurento media server with:

```
# Xenial
$ docker run -d --name kms -p 8888:8888 kurento/kurento-media-server:xenial-latest
# Trusty
$ docker run -d --name kms -p 8888:8888 kurento/kurento-media-server:trusty-latest
```

To check that kurento media server is ready and listening, issue the following command (you need to have curl installed on your system):

```
$ curl -i -N -H "Connection: Upgrade" -H "Upgrade: websocket" -H "Host: 127.0.0.1:8888
↪" -H "Origin: 127.0.0.1" http://127.0.0.1:8888/kurento
```

You will get something like:

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

Don't worry about the second line (`500 Internal Server Error`). It's ok, because we are not talking the protocol Kurento media server expects, we are just checking that the server is up and listening for connections.

1.2.4 Configuration

Kurento works by orchestrating a broad set of technologies that must be made to work together. Some of these technologies can accept different configuration parameters that Kurento makes available through several configuration files:

- `/etc/kurento/kurento.conf.json`: The main configuration file. Provides settings for the behavior of Kurento Media Server itself.

- `/etc/kurento/modules/kurento/MediaElement.conf.ini`: Generic parameters for all kinds of *MediaElement*.
- `/etc/kurento/modules/kurento/SdpEndpoint.conf.ini`: Audio/video parameters for *SdpEndpoint*s (i.e. *WebRtcEndpoint*and *RtpEndpoint*).
- `/etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini`: Specific parameters for *WebRtcEndpoint*.
- `/etc/kurento/modules/kurento/HttpEndpoint.conf.ini`: Specific parameters for *HttpEndpoint*.
- `/etc/default/kurento-media-server`: This file is loaded by the system's service init files. Defines some environment variables, which have an effect on features such as the *Debug Logging*, or the *Kernel Dump* files that are generated when a crash happens.

1.2.4.1 STUN and TURN Configuration

If Kurento Media Server is located behind a NAT you need to use a STUN or TURN in order to achieve NAT traversal. In most of cases, a STUN server will do the trick. A TURN server is only necessary when the NAT is symmetric.

The connection of these server is configured in the WebRtcEndpoint configuration file: `/etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini`

1.2.4.1.1 STUN Configuration

For configuring the STUN Server in Kurento you must *(uncomment and)* set the following parameters in the WebRtcEndPoint configuration file:

```
stunServerAddress=<stun_ip_address>
stunServerPort=<stun_port>
```

The parameter **stunServerAddress** should be an IP address (not domain name).

There is plenty of public STUN servers available, for example:

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

(continues on next page)

(continued from previous page)

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

1.2.4.1.2 TURN Configuration

For configuring the STUN Server in Kurento you must *(uncomment and)* set the following parameter in the WebRtcEndPoint configuration file:

```
turnURL=user:password@address:port
```

As before, TURN address should be an IP address (not domain name).

1.2.4.1.3 Remarks

1. Note that it is somewhat easy to find free STUN servers available on the net, because their functionality is pretty limited and it is not costly to keep them working for free. However, this doesn't happen with TURN servers, which act as a media proxy between peers and thus the cost of maintaining one is much higher. It is rare to find a TURN server which works for free while offering good performance. Usually, each user opts to maintain their own private TURN server instances.

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

3. In order to check the availability of either TURN and STUN servers you can check here: https://webrtc.github.io/samples/src/content/peerconnection/trickle-ice/

1.2.4.2 Debug Logging

Kurento Media Server generates log files that are stored in `/var/log/kurento-media-server/`. The content of this folder is as follows:

- `media-server_<timestamp>.<log_number>.<kms_pid>.log`: Output log of a currently running instance of KMS.
- `media-server_error.log`: Errors logged by third-party libraries.
- `logs`: Folder that contains older KMS logs. The logs in this folder are rotated, so they don't fill up all the space available in the disk.

Each line in a log produced by KMS has a fixed structure:

```
[timestamp] [pid] [memory] [level] [component] [filename:loc] [method] [message]
```

- `[timestamp]`: Date and time of the logging message (e.g. *2017-12-31 23:59:59,493295*).
- `[pid]`: Process Identifier of *kurento-media-sever* (e.g. *17521*).
- `[memory]`: Memory address in which the *kurento-media-sever* component is running (e.g. *0x00007fd59f2a78c0*).
- `[level]`: Logging level. This value typically will be *INFO* or *DEBUG*. If unexpected error situations happen, the *WARN* and *ERROR* levels will contain information about the problem.
- `[component]`: Name of the component that generated the log line. E.g. *KurentoModuleManager*, *webrtcendpoint*, or *qtmux*, among others.
- `[filename:loc]`: Source code file name (e.g. *main.cpp*) followed by the line of code number.
- `[method]`: Name of the function in which the log message was generated (e.g. *loadModule()*, *doGarbageCollection()*, etc).
- `[message]`: Specific log information.

For example, when KMS starts correctly, this trace is written in the log file:

```
[timestamp] [pid] [memory]  info  KurentoMediaServer  main.cpp:255  main()  Kurento␣
↪Media Server started
```

1.2.4.2.1 KMS Logging levels and components

Each different **component** of KMS is able to generate its own logging messages. Besides that, each individual logging message has a severity **level**, which defines how critical (or superfluous) the message is.

These are the different message levels, as defined by the GStreamer logging library:

- **(1) ERROR**: Logs all *fatal* errors. These are errors that do not allow the core or elements to perform the requested action. The application can still recover if programmed to handle the conditions that triggered the error.
- **(2) WARNING**: Logs all warnings. Typically these are *non-fatal*, but user-visible problems that *are expected to happen*.
- **(3) FIXME**: Logs all "fixme" messages. Fixme messages are messages that indicate that something in the executed code path is not fully implemented or handled yet. The purpose of this message is to make it easier to spot incomplete/unfinished pieces of code when reading the debug log.
- **(4) INFO**: Logs all informational messages. These are typically used for events in the system that *happen only once*, or are important and rare enough to be logged at this level.
- **(5) DEBUG**: Logs all debug messages. These are general debug messages for events that *happen only a limited number of times* during an object's lifetime; these include setup, teardown, change of parameters, etc.
- **(6) LOG**: Logs all log messages. These are messages for events that *happen repeatedly* during an object's lifetime; these include streaming and steady-state conditions.
- **(7) TRACE**: Logs all trace messages. These messages for events that *happen repeatedly* during an object's lifetime such as the ref/unref cycles.
- **(8) MEMDUMP**: Log all memory dump messages. Memory dump messages are used to log (small) chunks of data as memory dumps in the log. They will be displayed as hexdump with ASCII characters.

Logging categories and levels can be set by two methods:

- Use the specific command-line argument while launching KMS. For example, run:

```
/usr/bin/kurento-media-server \
  --gst-debug-level=3 \
  --gst-debug=Kurento*:4,kms*:4
```

- Use the environment variable GST_DEBUG. For example, run:

```
export GST_DEBUG="3,Kurento*:4,kms*:4"
/usr/bin/kurento-media-server
```

1.2.4.2.1.1 Suggested levels

Here are some tips on what logging components and levels could be most useful depending on what is the issue to be analyzed. They are given in the environment variable form, so they can be copied directly into the KMS configuration file, */etc/default/kurento-media-server*:

- Default suggested levels:

```
export GST_DEBUG="3,Kurento*:4,kms*:4"
```

- COMEDIA port discovery:

```
export GST_DEBUG="3,rtpendpoint:4"
```

- ICE candidate gathering:

```
export GST_DEBUG="3,kmsiceniceagent:5,kmswebrtcsession:5,webrtcendpoint:4"
```

  Notes:

  - *kmsiceniceagent* shows messages from the Nice Agent (handling of candidates).
  - *kmswebrtcsession* shows messages from the KMS WebRtcSession (decision logic).
  - *webrtcendpoint* shows messages from the WebRtcEndpoint (very basic logging).

- Event MediaFlow{In|Out} state changes:

```
export GST_DEBUG="3,KurentoMediaElementImpl:5"
```

- Player:

```
export GST_DEBUG="3,playerendpoint:5"
```

- Recorder:

```
export GST_DEBUG="3,KurentoRecorderEndpointImpl:4,recorderendpoint:5,qtmux:5"
```

- REMB congestion control:

```
export GST_DEBUG="3,kmsremb:5"
```

  Notes:

  - *kmsremb:5* (debug level 5) shows only effective REMB send/recv values.
  - *kmsremb:6* (debug level 6) shows full handling of all source SSRCs.

- RPC calls:

```
export GST_DEBUG="3,KurentoWebSocketTransport:5"
```

- RTP Sync:

```
export GST_DEBUG="3,kmsutils:5,rtpsynchronizer:5,rtpsynccontext:5,
↪basertpendpoint:5"
```

- SDP processing:

```
export GST_DEBUG="3,kmssdpsession:4"
```

- Transcoding of media:

```
export GST_DEBUG="3,Kurento*:5,kms*:4,agnosticbin*:7"
```

- Unit tests:

```
export GST_DEBUG="3,check:5"
```

1.2.4.2.1.2 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 has its own logging system that comes disabled by default, but can be enabled very easily. This can prove useful in situations where a developer is studying an issue with the ICE process. However, the debug output of libnice is very verbose, so it makes sense that it is left disabled by default for production systems.

Run KMS with these environment variables defined: `G_MESSAGES_DEBUG` and `NICE_DEBUG`. They must have one or more of these values, separated by commas:

- libnice
- libnice-stun
- libnice-tests
- libnice-socket
- libnice-pseudotcp
- libnice-pseudotcp-verbose
- all

Example:

```
export G_MESSAGES_DEBUG="libnice,libnice-stun"
export NICE_DEBUG="$G_MESSAGES_DEBUG"
/usr/bin/kurento-media-server
```

1.3 Programmers Manual

Welcome to Kurento's Programmer's Manual!

This User and Programmers Guide relates to the Stream Oriented GE which is part of the *Data/Context Management* chapter. Please find more information about this Generic Enabler in the following Open Specification.

Any feedback on this document is highly welcome, including bug reports, typos or stuff you think should be included but is not. Please send the feedback through Github. Thanks in advance!

1.3.1 Writing Kurento Applications

1.3.1.1 Global Architecture

Kurento can be used following the architectural principles of the web. That is, creating a multimedia application based on Kurento can be a similar experience to creating a web application using any of the popular web development frameworks.

At the highest abstraction level, web applications have an architecture comprised of three different layers:

- **Presentation layer (client side)**: Here we can find all the application code which is in charge of interacting with end users so that information is represented in a comprehensive way. This usually consists on HTML pages.
- **Application logic (server side)**: This layer is in charge of implementing the specific functions executed by the application.
- **Service layer (server or Internet side)**: This layer provides capabilities used by the application logic such as databases, communications, security, etc. These services can be hosted in the same server as the application logic, or can be provided by external parties.

Following this parallelism, multimedia applications created using Kurento can also be implemented with the same architecture:

- **Presentation layer (client side)**: Is in charge of multimedia representation and multimedia capture. It is usually based on specific built-in capabilities of the client. For example, when creating a browser-based application, the presentation layer will use capabilities such as the `<video>` HTML tag or the *WebRTC* JavaScript APIs.
- **Application logic**: This layer provides the specific multimedia logic. In other words, this layer is in charge of building the appropriate pipeline (by chaining the desired Media Elements) that the multimedia flows involved in the application will need to traverse.
- **Service layer**: This layer provides the multimedia services that support the application logic such as media recording, media ciphering, etc. The Kurento Media Server (i.e. the specific *Media Pipeline* of *Media Elements*) is in charge of this layer.

The interesting aspect of this discussion is that, as happens with web development, Kurento applications can place the Presentation layer at the client side and the Service layer at the server side. However the Application logic, in both cases, can be located at either of the sides or even distributed between them. This idea is represented in the following picture:

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**.

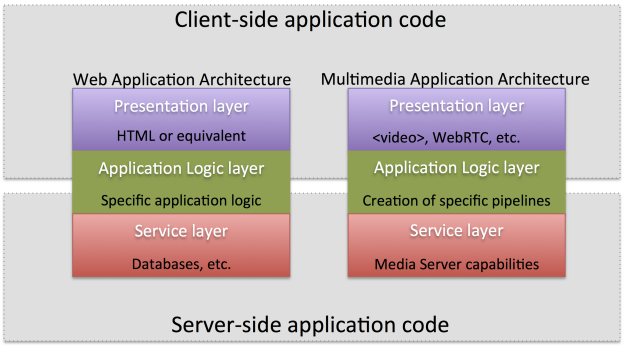

Fig. 1: *Layered architecture of web and multimedia applications. Applications created using Kurento (right) can be similar to standard Web applications (left). Both types of applications may choose to place the application logic at the client or at the server code.*

1.3.1.2 Application Architecture

Kurento, as most multimedia communication technologies out there, is built using two layers (called *Planes*) to abstract key functions in all interactive communication systems:

- **Signaling Plane**. The parts of the system in charge of the management of communications, that is, the modules that provides functions for media negotiation, QoS parametrization, call establishment, user registration, user presence, etc. are conceived as forming part of the *Signaling Plane*.
- **Media Plane**. Functionalities such as media transport, media encoding/decoding and media processing make the *Media Plane*, which takes care of handling the media. The distinction comes from the telephony differentiation between the handling of voice and the handling of meta-information such as tone, billing, etc.

The following figure shows a conceptual representation of the high level architecture of Kurento:

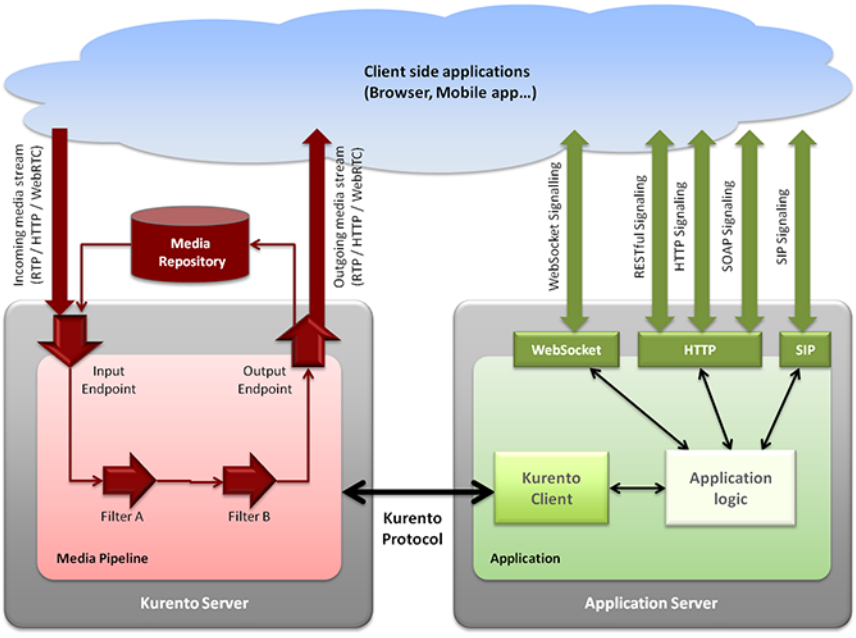

Fig. 2: *Kurento Architecture. Kurento architecture follows the traditional separation between signaling and media Planes.*

The **right side** of the picture shows the application, which is in charge of the signaling Plane and contains the business

logic and connectors of the particular multimedia application being deployed. It can be build with any programming technology like Java, Node.js, PHP, Ruby, .NET, etc. The application can use mature technologies such as *HTTP* and *SIP* Servlets, Web Services, database connectors, messaging services, etc. Thanks to this, this Plane provides access to the multimedia signaling protocols commonly used by end-clients such as *SIP*, RESTful and raw HTTP based formats, SOAP, RMI, CORBA or JMS. These signaling protocols are used by client side of applications to command the creation of media sessions and to negotiate their desired characteristics on their behalf. Hence, this is the part of the architecture, which is in contact with application developers and, for this reason, it needs to be designed pursuing simplicity and flexibility.

On the **left side**, we have the Kurento Media Server, which implements the media Plane capabilities providing access to the low-level media features: media transport, media encoding/decoding, media transcoding, media mixing, media processing, etc. The Kurento Media Server must be capable of managing the multimedia streams with minimal latency and maximum throughput. Hence the Kurento Media Server must be optimized for efficiency.

1.3.1.2.1 Communicating client, server and Kurento

As can be observed in the figure above, a Kurento application involves interactions among three main modules:

- **Client Application**: Involves the native multimedia capabilities of the client platform plus the specific client-side application logic. It can use Kurento Clients designed for client platforms (for example, Kurento JavaScript Client).
- **Application Server**: Involves an application server and the server-side application logic. It can use Kurento Clients designed 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:

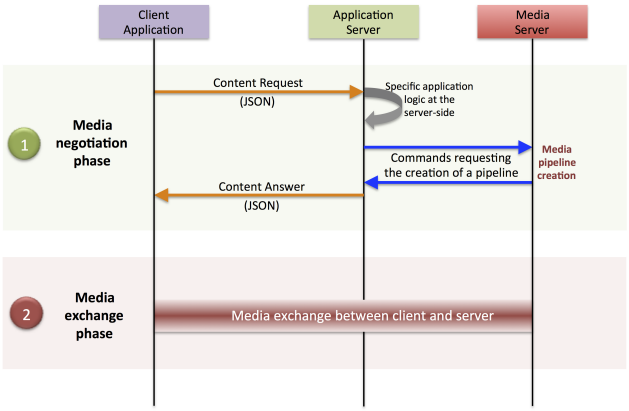

Fig. 3: *Main interactions between architectural modules. These occur in two phases: negotiation and media exchange. Remark that the color of the different arrows and boxes is aligned with the architectural figures presented above. For example, orange arrows show exchanges belonging to the signaling Pipeline, blue arrows show exchanges belonging to the Kurento Protocol, red boxes are associated to the Kurento Media Server, and green boxes with the application.*

1.3.1.2.1.1 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.

1.3.1.2.1.2 2. Media exchange phase

After the signaling part, a new phase starts with the aim to produce the actual media exchange. The client addresses a request for the media to the Kurento Media Server using the information gathered during the negotiation phase.

Following with the video-clip visualization example mentioned above, the browser will send a GET request to the IP address and port of the Kurento Media Server where the clip can be obtained and, as a result, an HTTP reponse containing the media will be received.

Following the discussion with that simple example, one may wonder why such a complex scheme for just playing a video, when in most usual scenarios clients just send the request to the appropriate URL of the video without requiring any negotiation. The answer is straightforward. Kurento is designed for media applications involving complex media processing. For this reason, we need to establish a two-phase mechanism enabling a negotiation before the media exchange. The price to pay is that simple applications, such as one just downloading a video, also need to get through these phases. However, the advantage is that when creating more advanced services the same simple philosophy will hold. For example, if we want to add Augmented Reality or Computer Vision features to that video-clip, we just need to create the appropriate pipeline holding the desired Media Elements during the negotiation phase. After that, from the client perspective, the processed clip will be received as any other video.

1.3.1.2.2 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:

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.

1.3.1.3 Media Pipeline

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

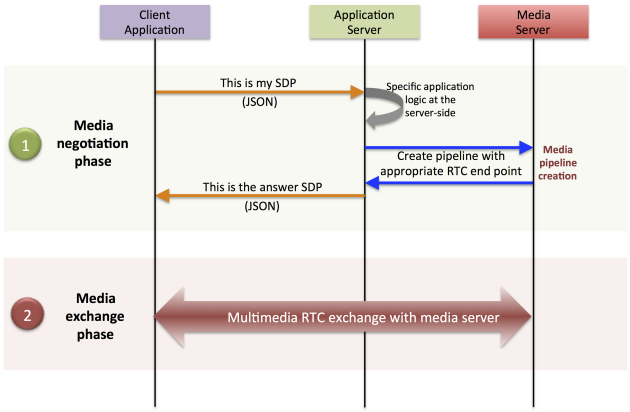

Fig. 4: *Interactions in a WebRTC session. During the negotiation phase, an SDP Offer is sent to KMS, requesting the capabilities of the client. As a result, Kurento Media Server generates an SDP Answer that can be used by the client for establishing the media exchange.*

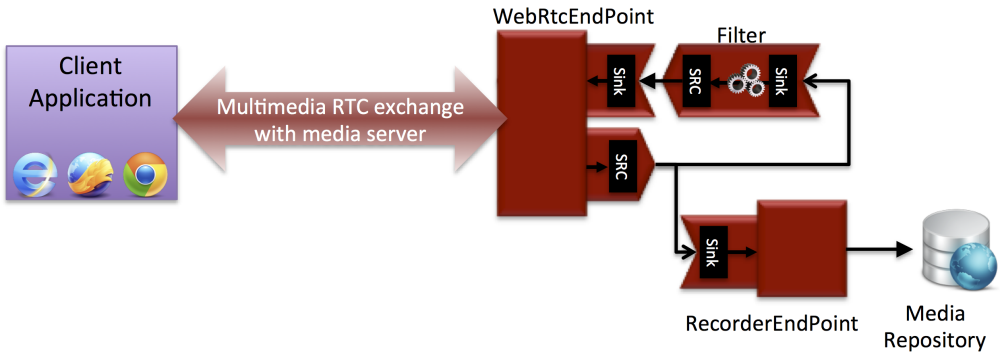

Fig. 5: *Example pipeline for a WebRTC session. A WebRtcEndpoint is connected to a RecorderEndpoint storing the received media stream and to an Augmented Reality filter, which feeds its output media stream back to the client. As a result, the end user will receive its own image filtered (e.g. with a hat added onto her head) and the stream will be recorded and made available for further recovery into a repository (e.g. a file).*

capabilities they want to use (the Media Elements) and the topology determining which Media Element provides media to which other Media Elements (the connectivity).

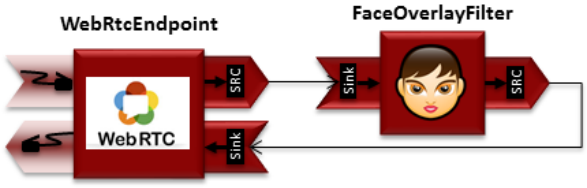

Fig. 6: *Simple Example of a Media Pipeline*

The connectivity is controlled through the *connect* primitive, exposed on all Kurento Client APIs.

This primitive is always invoked in the element acting as source and takes as argument the sink element following this scheme:

```
sourceMediaElement.connect(sinkMediaElement)
```

For example, if you want to create an application recording WebRTC streams into the file system, you'll need two media elements: *WebRtcEndpoint* and *RecorderEndpoint*. When a client connects to the application, you will need to instantiate these media elements making the stream received by the *WebRtcEndpoint* (which is capable of receiving WebRTC streams) to be fed to the *RecorderEndpoint* (which is capable of recording media streams into the file system). Finally you will need to connect them so that the stream received by the former is transferred into the later:

```
WebRtcEndpoint.connect(RecorderEndpoint)
```

To simplify the handling of WebRTC streams in the client-side, Kurento provides an utility called *WebRtcPeer*. Nevertheless, the standard WebRTC API (*getUserMedia*, *RTCPeerConnection*, and so on) can also be used to connect to *WebRtcEndpoints*. For further information please visit the *Tutorials section*.

1.3.2 Integration with Orion: kurento-fiware java module

The Kurento team has developed a small module that you can use in your own application and it would make it easier to connect with Orion, and make use of it in order to gather the context generated by your application related with kurento (MediaEvents and Devices). Also it has been designed to be easily extended so you can implement your own publishers and readers for any other entity.

1.3.2.1 How to use it

1. Clone the gitHub repository.

   ```
   git clone https://github.com/Kurento/kurento-fiware-java.git
   ```

2. Go to the project kurento-fiware *(the other folder contains an example of how to use it)*.

   ```
   cd kurento-fiware
   ```

3. Build and install in your maven local repo *(you will need Java >= 1.8)*.

   ```
   mvn install
   ```

4. Import in your project the dependency

```
<dependency>
    <groupId>org.kurento</groupId>
    <artifactId>kurento-fiware</artifactId>
    <version>2.0-SNAPSHOT</version>
</dependency>
```

1.3.2.2 Processing Media Streams

Any Media Stream that flows in Kurento can be processed by one or more modules. Some modules would rise Media Events following the logic they are meant to with associated information. For example, the built-in module "kms-platedetector" would rise a Media Event whenever a traffic plate is detected, this event would contain the plate number read from the Media Stream. This Events should be collected by the client application an the application should act upon them.

1.3.2.2.1 Kurento Events

You can configure to receive Kurento Events into your client application and extract relevant context information that can be fed to Orion.

Example:

You develop a Client Application with a Kurento attached and the kms-platedetector built-in module. Each time a traffic plate is detected , Kurento rises a Event and your application reacts by:

- Inserting the Event as a MediaEvent in Orion.
- Looking for the plate in Orion, and update the location of the vehicle associated.

1.3.2.2.2 MediaEvents to Orion

MediaEvent is a generic FIWARE DataModel *(still under approval process)* that allows any application using Kurento to insert in Orion any type or Kurento Event risen by any module *If you are using Java for implementing your client application you can use* kurento-fiware *module to simplify all the integration*.

While using the provided library you need just to:

1. Configure the connection to the Orion instance by:

```
final OrionConnectorConfiguration orionConnectorConfiguration = new␣
↪OrionConnectorConfiguration();
```

By default the configuration points to http://localhost:1026

2. Define how to exactly match the detected Event to the MediaEvent DataModel extending the `MediaEventPublisher` defining a `mapEntityToOrionEntity`. E.g.

```
public MediaEvent mapEntityToOrionEntity(DevicePlateDetectedEvent kurentoEvent) {
  MediaEvent orion_entity = new MediaEvent();
  orion_entity.setId(...);
  orion_entity._getGsmaCommons().setDateCreated(kurentoEvent.getTimestamp());
```

(continues on next page)

(continued from previous page)

```
  orion_entity.setData(kurentoEvent.getPlate());
  if (kurentoEvent.getCamera() != null) {
    orion_entity.setDeviceSource(kurentoEvent.getCamera().getId());
  }
  orion_entity.setMediasource(mapKurentoMediaSource(kurentoEvent.getSource()));
  return orion_entity;
}
```

3. And publish the event.

```
plateDetectedEventPublisher.publish(extendedEvent);
```

You can check the specifics of the MediaEvent DataModel here.

1.3.2.3 Devices

The cameras used for generating the Media Streams are also part of the context so we expect them to be also part of the information that can be found in Orion. Devices are an an integral part of the common entities found in Orion and they are defined as a very generic FIWARE DataModel, so any kind of "camera" used in the Kurento Client application can be represented in this DataModel.

If your Client Kurento Application is developed in Java you can also make use of the provided kurento-fiware module to simplify all the integration.

The essential steps for inserting Devices into Orion are similar to the Media Events' ones:

1. Configure the connection to the Orion instance by:

```
final OrionConnectorConfiguration orionConnectorConfiguration = new␣
↪OrionConnectorConfiguration();
```

By default the configuration points to http://localhost:1026

2. Define how to exactly match the custom Camera used in the application to the Device DataModel extending the `DevicePublisher` defining a `mapEntityToOrionEntity`. E.g.

```
public Device mapEntityToOrionEntity(Camera cam) {

  String[] supportedProtocol = { "WebRTC" };

  Device entity = new Device();

  entity.setControlledAsset(cam.getControlledAsset());
  entity.setDateInstalled(cam.getCreationDate());
  entity.setDeviceState(cam.getState());
  entity._getDeviceCommons().setSupportedProtocol(supportedProtocol);
  entity._getGsmaCommons().setId(cam.getId());
  entity._getGsmaCommons().setDateCreated(cam.getCreationDate());
  entity._getGsmaCommons().setDescription("Plate detector camera example");
  entity._getGsmaCommons().setName(cam.getName());
  entity.setIpAddress(cam.getIp());
```

(continues on next page)

(continued from previous page)

```
  return entity;
}
```

3. Publish the Device.

```
CamPublisher cameraPublisher = new CamPublisher(orionConnectorConfiguration);
cameraPublisher.publish(cam);
```

4. Update the Device for each change of state (e.g. "PAUSED" / "PROCESSING") or each last value detected.

```
final OrionConnectorConfiguration orionConnectorConfiguration = new␣
↪OrionConnectorConfiguration();
CamPublisher cameraPublisher = new CamPublisher(orionConnectorConfiguration);
CamReader cameraReader = new CamReader(orionConnectorConfiguration);
Camera cam = cameraReader.readObject(id);
/* Update values of cam */
cameraPubliser.update(cam);
```

1.3.2.4 Other entities

While developing your Smart Solution you would need to work with other Entities in Orion, for example Vehicles, Alerts, places such as museums, gardens, etc. While Kurento Entities aren't directly related to these, the kurento-fiware module, provides an easy way of extending its functionality to any other DataModel and any other custom Object.

In this case you can replicate the structure that the module provides for the Device and MediaEvent entities in your project. This means to provide the following classes:

- **YourOrionEntity.java:** That is the class to map the OrionEntity you need. This class must implement the OrionEntity Interface.
- **<YourOrionEntity>OrionPublisher.java:** This needs to be an extension of the DefaultOrionPublisher. You will need to define:
  - `O`: the OrionEntity that will be published in orion in this case <*YourOrionEntity*>.
  - `T`: a custom class that can be mapped to <*YourOrionEntity*>.
  - `mapEntityToOrionEntity` is the method that would be able to map from `T` to `O`.
- **<YourOrionEntity>OrionReader.java:** This needs to be an extension of the DefaultOrionReader. As for the publisher you will need to define:
  - `O`: the OrionEntity that will be published in orion in this case <*YourOrionEntity*>.
  - `T`: a custom class that can be mapped to <*YourOrionEntity*>.
  - `mapOrionEntityToEntity`; is the method that would be able to map from `O` to `P`.

In case *YourOrionEntity* needs some processing for presenting a plain JSON to Orion or for reading it you may need to provide also a **<YourOrionEntity>JsonManager** to the OrionConnector.

1.3.2.5 More

Follow the links for more information about the kurento-fiware module:

1.3.2.5.1 Java Module - Plate Detector Filter

This web application consists on a *WebRTC* video communication in mirror (*loopback*) with a plate detector filter element.

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

1.3.2.5.1.1 For the impatient: running this example

First of all, you should have available:

- An instance of kurento running with the kms-platedetector module. Further information on the installation guide.
- An instance of orion running. See: Orion installation guide.

To launch the application, you need:

1. To clone the GitHub project where this demo is hosted:

```
git clone https://github.com/Kurento/kurento-fiware-java
```

2. Install the kurento-fiware module:

```
cd kurento-fiware-java/kurento-fiware
mvn install
```

3. Run the application

```
cd ../kurento-tutorial-java/kurento-platedetector-fiware
mvn -U clean spring-boot:run -Dkms.url=ws://localhost:8888/kurento
```

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 both Kurento Media Server and Orion are up and running in the same machine as the tutorial.

1.3.2.5.1.2 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 a HTML5 video tag that is activated once the camera is registered in orion. The video camera stream (the local client-side stream) is sent to Kurento Media Server, which processes it and registers the events rised in Orion. 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 Plate Detector Module Tutorial. A screenshot of the running example is shown in the following picture:

Fig. 7: *WebRTC with plateDetector filter Media Pipeline*

Fig. 8: *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 and the event is registered in Orion. As shown in the screenshot above, 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>() {

             @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();
```

(continues on next page)

(continued from previous page)

```
        webRtcEndpoint.connect(plateDetectorFilter);
        plateDetectorFilter.connect(webRtcEndpoint);

        plateDetectorFilter.addPlateDetectedListener(new EventListener
↪<PlateDetectedEvent>() {
              @Override
              public void onEvent(PlateDetectedEvent event) {

                final OrionConnectorConfiguration orionConnectorConfiguration = new␣
↪OrionConnectorConfiguration();

                final PlateDetectedEventPublisher plateDetectedEventPublisher = new␣
↪PlateDetectedEventPublisher(
                    orionConnectorConfiguration);

                DevicePlateDetectedEvent extendedEvent = new␣
↪DevicePlateDetectedEvent(event, null);

                // TODO add the camera information (from {@link: CameraSession}

                JsonObject response = new JsonObject();
                response.addProperty("id", "plateDetected");
                response.addProperty("plate", event.getPlate());
                log.debug("plateDetectorFilter.onEvent({}) => {}", event.getPlate(),␣
↪response.toString());
                try {
                      session.sendMessage(new TextMessage(response.toString()));
                      plateDetectedEventPublisher.publish(extendedEvent);
                      log.debug("");
                } catch (OrionConnectorException e) {
                      log.warn("Could not publish event in ORION");
                      sendError(session, e.getMessage());
                } catch (Throwable t) {
                      log.warn("Thowable: {}", t.getLocalizedMessage());
                      sendError(session, t.getMessage());
                }
              }
        });

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

        // Sending response back to client
        JsonObject response = new JsonObject();
        response.addProperty("id", "startResponse");
        response.addProperty("sdpAnswer", sdpAnswer);

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

        webRtcEndpoint.gatherCandidates();

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

(continues on next page)

(continued from previous page)

```
        }
}
```

1.3.2.5.1.3 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 four dependencies: the Kurento Client Java dependency (*kurento-client*) and the JavaScript Kurento utility library (*kurento-utils*) for the client-side, the KMS platedetector module (*platedetector*) and the kurento-fiware module (*kurento-fiware*).

1.3.3 Writing Kurento Modules

You can expand the Kurento Media Server developing your own modules. There are two flavors of Kurento modules:

- Modules based on *OpenCV*. This kind of modules are recommended if you would like to develop a filter providing Computer Vision or Augmented Reality features.
- Modules based on *GStreamer*. This kind of modules provide a generic entry point for media processing with the GStreamer framework. Such modules are more powerful but also they are more difficult to develop. Skills in GStreamer development are necessary.

The starting point to develop a filter is to create the filter structure. For this task, you can use the `kurento-module-scaffold` tool. This tool is distributed with the `kurento-media-server-dev` package. To install this tool run this command:

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

The tool usage is different depending on the chosen flavor:

1. OpenCV module:

   ```
   kurento-module-scaffold.sh <module_name> <output_directory> opencv_filter
   ```

2. Gstreamer module:

   ```
   kurento-module-scaffold.sh <module_name> <output_directory>
   ```

The tool generates the folder tree, all the needed `CmakeLists.txt` files, and example files of Kurento module descriptor files (.kmd). These files contain the description of the modules, the constructor, the methods, the properties, the events and the complex types defined by the developer.

Once *kmd* files are completed it is time to generate the corresponding code. The tool `kurento-module-creator` generates glue code to server-side. Run this from the root directory:

```
cd build
cmake ..
```

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

1.3.3.1 OpenCV module

We have four files in `src/server/implementation/`:

```
ModuleNameImpl.cpp
ModuleNameImpl.hpp
ModuleNameOpenCVImpl.cpp
ModuleNameOpenCVImpl.hpp
```

The first two files should not be modified. The last two files will contain the logic of your module.

The file `ModuleNameOpenCVImpl.cpp` contains functions to deal with the methods and the parameters (you must implement the logic). Also, this file contains a function called `process`. This function will be called with each new frame, thus you must implement the logic of your filter inside it.

1.3.3.2 GStreamer module

In this case, we have two directories inside the `src/` folder:

- The `gst-plugins/` folder contains the implementation of your GStreamer Element (the `kurento-module-scaffold` generates a dummy filter).
- Inside the `server/objects/` folder you have two files:

```
ModuleNameImpl.cpp
ModuleNameImpl.hpp
```

In the file `ModuleNameImpl.cpp` you have to invoke the methods of your GStreamer element. The module logic will be implemented in the GStreamer Element.

1.3.3.3 For both kind of modules

If you need extra compilation dependencies you can add compilation rules to the *kurento-module-creator* using the function `generate_code` in the `CmakeLists.txt` file, located in `src/server/`.

The following parameters are available:

- `SERVER_STUB_DESTINATION` (required) The generated code that you may need to modify will be generated on the folder indicated by this parameter.
- `MODELS` (required) This parameter receives the folders where the models (.kmd files) are located.
- `INTERFACE_LIB_EXTRA_SOURCES` `INTERFACE_LIB_EXTRA_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[ libversion range]` (possible ranges can use symbols `AND` `OR` < <= > >= ^ and ~).
  - ^ indicates a version compatible using *Semantic Versioning*.

  - ~ Indicates a version similar, that can change just last indicated version character.

Once the module logic is implemented and the compilation process is finished, you need to install your module in your system. You can follow two different ways:

1. You can generate the Debian package (`debuild -us -uc`) and install it (`dpkg -i`). 2. You can define the following environment variables in the file `/etc/default/kurento`:

```
KURENTO_MODULES_PATH=<module_path>/build/src
GST_PLUGIN_PATH=<module_path>/build/src
```

Now, you need to generate code for Java or JavaScript to use your module from the client-side.

- For Java, from the build directory you have to execute `cmake .. -DGENERATE_JAVA_CLIENT_PROJECT=TRUE` command, that generates a Java folder with client code. You can run `make java_install` and your module will be installed in your Maven local repository. To use the module in your Maven project, you have to add the dependency to the `pom.xml` file:

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

- For JavaScript, you should run `cmake .. -DGENERATE_JS_CLIENT_PROJECT=TRUE`. This command generates a `js/` folder with client code. Now you can manually add the JavaScript library to use your module in your application. Alternatively, you can use *Bower* (for *Browser JavaScript*) or *NPM* (for *Node.js*). To do that, you should add your JavaScript module as a dependency in your `bower.json` or `package.json` file respectively, as follows:

```
"dependencies": {
  "modulename": "moduleversion"
}
```

1.3.3.4 Examples

Simple examples for both kind of modules are available in GitHub:

- OpenCV module.
- GStreamer module.

There are a lot of examples showking how to define methods, parameters or events in all our public built-in modules:

- kms-pointerdetector.
- kms-crowddetector.
- kms-chroma.
- kms-platedetector.

Moreover, all our modules are developed using this methodology. For that reason you can take a look to our main modules:

- kms-core.
- kms-elements.
- kms-filters.

1.3.4 Tutorials

Here you can check the different tutorials/examples available to comprehend Kurento and start using it for your benefit.

1.3.4.1 Genal Java Kurento tutorials

1.3.4.2 Kurento modules Java tutorials

1.3.4.3 Smart Solition tutorials

1.4 FIWARE Stream Oriented Generic Enabler - Open API Specification

The Stream Oriented API is a resource-oriented API accessed via WebSockets that uses JSON-RPC V2.0 based representations for information exchange. An RPC call is represented by sending a **request** message to a server. Each request message has the following members:

- *jsonrpc*: a string specifying the version of the JSON-RPC protocol. It must be exactly *2.0*.
- *id*: an 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.

When an RPC call is made by a client, the server replies with a **response** object. In the case of a success, the response object contains the following members:

- *jsonrpc*: it must be exactly *2.0*.
- *id*: it must match the value of the id member in the request object.
- *result*: structured value which contains the invocation result.

In the case of an **error**, the response object contains the following members:

- *jsonrpc*: it must be exactly *2.0*.
- *id*: it must match the value of the id member in the request object.
- *error*: object describing the error through the following members:
  - *code*: integer number that indicates the error type that occurred
  - *message*: string providing a short description of the error.
  - *data*: 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.

Therefore, the value of the *method* parameter in the request determines the type of request/response to be exchanged between client and server. The following section describes each pair of messages depending of the type of *method* (namely: *Ping*, *Create*, *Invoke*, *Release*, *Subscribe*, *Unsubscribe*, and *OnEvent*).

1.4.1 Ping

In order to warranty 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 is aware that the connectivity with the media server has been lost.

1.4.1.1 Request

A *ping* request contains the following parameters:

- *method* (required, string). Value: *ping*.
- *params* (required, object). Parameters for the invocation of the ping message, containing these member:
  - interval (required, number). Time out to receive the *pong* message from the server, in milliseconds. By default this value is *240000* (i.e. 40 seconds).

This is an example of *ping*:

- Body (application/json)

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

1.4.1.2 Response

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

- Body (application/json)

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

1.4.2 Create

Create message requests the creation of an Media Pipelines and Media Elements in the Media Server. The parameter type specifies the type of the object to be created. The parameter *params* contains all the information needed to create the object. Each message needs different parameters to create the object.

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.

1.4.2.1 Request

A *create* request contains the following parameters:

- *method* (required, string). Value: *create*.
- *params* (required, object). Parameters for the invocation of the create message, containing these members:
  - type (required, string). Media pipeline or media element to be created. The allowed values are the following:
    - *MediaPipeline*: Media Pipeline to be created.
    - *WebRtcEndpoint*: This media element offers media streaming using WebRTC.
    - *RtpEndpoint*: Media element that provides bidirectional content delivery capabilities with remote networked peers through RTP protocol. It contains paired sink and source MediaPad for audio and video.
    - *HttpPostEndpoint*: This type of media element provides unidirectional communications. Its MediaSource are related to HTTP POST method. It contains sink MediaPad for audio and video, which provide access to an HTTP file upload function.
    - *PlayerEndpoint*: It provides function to retrieve contents from seekable sources in reliable mode (does not discard media information) and inject them into KMS. It contains one MediaSource for each media type detected.
    - *RecorderEndpoint*: Provides function to store contents in reliable mode (doesn't discard data). It contains MediaSink pads for audio and video.
    - *FaceOverlayFilter*: It detects faces in a video feed. The face is then overlaid with an image.
    - *ZBarFilter*: This Filter detects QR and bar codes in a video feed. When a code is found, the filter raises a CodeFound.
    - *GStreamerFilter*: This is a generic Filter interface, that creates GStreamer filters in the media server.
    - *Composite*: A Hub that mixes the audio stream of its connected sources and constructs a grid with the video streams of its connected sources into its sink.
    - *Dispatcher*: A Hub that allows routing between arbitrary port pairs.
    - *DispatcherOneToMany*: A Hub that sends a given source to all the connected sinks.
  - *constructorParams* (required, object). Additional parameters. For example:
    - *mediaPipeline* (optional, string): This parameter is only mandatory for Media Elements. In that case, the value of this parameter is the identifier of the media pipeline which is going to contain the Media Element to be created.
    - *uri* (optional, string): This parameter is only required for Media Elements such as *PlayerEndpoint* or *RecorderEndpoint*. It is an URI used in the Media Element, i.e. the media to be played (for *PlayerEndpoint*) or the location of the recording (for *RecorderEndpoint*).
    - *properties* (optional, object): Array of additional objects (key/value).
  - *sessionId* (optional, string). Session identifier. This parameter is not present in the first request (typically the media pipeline creation).

The following example shows a request message requesting the creation of an object of the type *MediaPipeline*:

- Body (application/json)

```
{
    "id": 2,
    "method": "create",
    "params": {
        "type": "MediaPipeline",
```

(continues on next page)

(continued from previous page)

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

The following example shows a request message requesting 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 was unknown for the client):

- Body (application/json)

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

1.4.2.2 Response

The response message contains the identifier of the new object in the field value. As usual, the message *id* must match with the request message. The *sessionId* is also returned in each response. A *create* response contains the following parameters:

- *result* (required, object). Result of the create invocation:
  - *value* (required, number). Identifier of the created media element.
  - *sessionId* (required, string). Session identifier.

The following examples shows the responses to the previous request messages (respectively, the response to the *MediaPipeline* create message, and then the response to the to *WebRtcEndpoint* create message). In the first example, the parameter `value` identifies the created Media Pipelines, and `sessionId` is the identifier of the current session.

- Body (application/json)

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

In the second response example, the parameter `value` identifies the created Media Element (a `WebRtcEndpoint` in this case). Notice that this value also identifies the Media Pipeline in which the Media Element is contained. The parameter `sessionId` is also contained in the response.

- Body (application/json)

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

1.4.3 Invoke

Invoke message requests the invocation of an operation in the specified object. The parameter object indicates the identifier 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* contains the parameters needed to execute the operation.

1.4.3.1 Request

An *invoke* request contains the following parameters:

- *method* (required, string). Value is *invoke*.
- *params* (required, object)
  - *object* (required, number). Identifier of the source media element.
  - *operation* (required, string). Operation invoked. Allowed Values:
    - *connect*. Connect two media elements.
    - *play*. Start the play of a media (*PlayerEndpoint*).
    - *record*. Start the record of a media (*RecorderEndpoint*).
    - *setOverlayedImage*. Set the image that is going to be overlaid on the detected faces in a media stream (*FaceOverlayFilter*).
    - *processOffer*. Process the offer in the SDP negotiation (*WebRtcEndpoint*).
    - *gatherCandidates*. Start the ICE candidates gathering to establish a WebRTC media session (*WebRtcEndpoint*).
    - *addIceCandidate*. Add ICE candidate (*WebRtcEndpoint*).
  - *operationParams* (optional, object).
    - *sink* (required, number). Identifier of the sink media element.
    - *offer* (optional, string). SDP offer used in the WebRTC SDP negotiation (in *WebRtcEndpoint*).
  - *sessionId* (required, string). Session identifier.

The following example shows a request message requesting the invocation of the operation connect on a *PlayerEndpoint* connected to a *WebRtcEndpoint*:

- Body (application/json)

```
{
    "id": 5,
    "method": "invoke",
    "params": {
        "object": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline/
↪76dcb8d7-5655-445b-8cb7-cf5dc91643bc_kurento.PlayerEndpoint",
        "operation": "connect",
        "operationParams": {
            "sink": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline/
↪087b7777-aab5-4787-816f-f0de19e5b1d9_kurento.WebRtcEndpoint"
        },
        "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
    },
    "jsonrpc": "2.0"
}
```

1.4.3.2 Response

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.

An *invoke* response contains the following parameters:

- *result* (required, object)
  - *sessionId* (required, string). Session identifier.
  - *value* (optional, object). Additional object which describes the result of the *Invoke* operation. For example, in a *WebRtcEndpoint* this field is the SDP response (WebRTC SDP negotiation).

The following example shows a typical response while invoking the operation connect:

- Body (application/json)

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

1.4.4 Release

Release message requests the release of the specified object. The parameter *object* indicates the id of the object to be released:

1.4.4.1 Request

A *release* request contains the following parameters:

- *method* (required, string). Value is *release*.
- *params* (required, object).
  - *object* (required, number). Identifier of the media element or pipeline to be released.

  - *sessionId* (required, string). Session identifier.

- Body (application/json)

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

1.4.4.2 Response

A *release* response contains the following parameters:

- *result* (required, object)
  - *sessionId* (required, string). Session identifier.

The response message only contains the *sessionId*. The following example shows the typical response of a release request:

- Body (application/json)

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

1.4.5 Subscribe

Subscribe 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.

1.4.5.1 Request

A *subscribe* request contains the following parameters:

- *method* (required, string). Value is *subscribe*.
- *params* (required, object). Parameters for the invocation of the subscribe message, containing these members:
  - *type* (required, string). Media event to be subscribed. The allowed values are the following:
    - *CodeFoundEvent*: raised by a *ZBarFilter* when a code is found in the data being streamed.
    - *ConnectionStateChanged*: Indicates that the state of the connection has changed.
    - *ElementConnected*: Indicates that an element has been connected to other.

        * *ElementDisconnected*: Indicates that an element has been disconnected.
        * *EndOfStream*: Event raised when the stream that the element sends out is finished.
        * *Error*: An error related to the MediaObject has occurred.
        * *MediaSessionStarted*: Event raised when a session starts. This event has no data.
        * *MediaSessionTerminated*: Event raised when a session is terminated. This event has no data.
        * *MediaStateChanged*: Indicates that the state of the media has changed.
        * *ObjectCreated*: Indicates that an object has been created on the media server.
        * *ObjectDestroyed*: Indicates that an object has been destroyed on the media server.
        * *OnIceCandidate*: Notify of a new gathered local candidate.
        * *OnIceComponentStateChanged*: Notify about the change of an ICE component state.
        * *OnIceGatheringDone*: Notify that all candidates have been gathered.
    - *object* (required, string). Media element identifier in which the event is subscribed.
    - *sessionId* (required, string). Session identifier.

The following example shows a request message requesting the subscription of the event type *EndOfStream* on a *PlayerEndpoint* Media Element:

- Body (application/json)

```
{
    "id": 11,
    "method": "subscribe",
    "params": {
        "type": "EndOfStream",
        "object": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline/
↪76dcb8d7-5655-445b-8cb7-cf5dc91643bc_kurento.PlayerEndpoint",
        "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
    },
    "jsonrpc": "2.0"
}
```

1.4.5.2 Response

The response message contains the subscription identifier. This value can be used later to remove this *subscription*.

A *subscribe* response contains the following parameters:

- *result* (required, object). Result of the subscription invocation. This object contains the following members:
    - *value* (required, number). Identifier of the media event.
    - *sessionId* (required, string). Session identifier.

The following example shows the response of subscription request. The *value* attribute contains the subscription identifier:

- Body (application/json)

```
{
    "id": 11,
    "result": {
        "value": "052061c1-0d87-4fbd-9cc9-66b57c3e1280",
```

(continues on next page)

(continued from previous page)

```
        "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
    },
    "jsonrpc": "2.0"
}
```

1.4.6 Unsubscribe

Unsubscribe 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.

1.4.6.1 Request

An *unsubscribe* request contains the following parameters:

- *method* (required, string). Value is *unsubscribe*.
- *params* (required, object).
  - *object* (required, string). Media element in which the subscription is placed.
  - *subscription* (required, number). Subscription identifier.
  - *sessionId* (required, string). Session identifier.

The following example shows a request message requesting the cancellation of the *subscription 353be312-b7f1-4768-9117-5c2f5a087429*:

- Body (application/json)

```
{
    "id": 38,
    "method": "unsubscribe",
    "params": {
        "subscription": "052061c1-0d87-4fbd-9cc9-66b57c3e1280",
        "object": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline/
↪76dcb8d7-5655-445b-8cb7-cf5dc91643bc_kurento.PlayerEndpoint",
        "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
    },
    "jsonrpc": "2.0"
}
```

1.4.6.2 Response

The response message only contains the *sessionId*. The following example shows the typical response of an unsubscription request:

An *unsubscribe* response contains the following parameters:

- *result* (required, object)
  - *sessionId* (required, string). Session identifier.

For example:

- Body (application/json)

```
{
    "id": 38,
    "result": {
        "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
    },
    "jsonrpc": "2.0"
}
```

1.4.7 OnEvent

When a client is *subscribed* to a type of events in 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 Stream Oriented open API is implemented with WebSockets and there is a full duplex channel between client and server.

1.4.7.1 Request

An *OnEvent* request contains the following parameters:

- *method* (required, string). Value is *onEvent*.
- *params* (required, object).
  - *value* (required, object)
    - *data* (required, object)
      - *source* (required, string). Source media element.
      - *tags* (optional, string array). Metadata for the media element.
      - *timestamp* (required, number). Media server time and date (in Unix time, i.e., number of seconds since 01/01/1970).
      - *type* (required, string). Same type identifier described on *subscribe* message (i.e.: *CodeFound*, *ConnectionStateChanged*, *ElementConnected*, *ElementDisconnected*, *EndOfStream*, *Error*, *MediaSessionStarted*, *MediaSessionTerminated*, *MediaStateChanged*, *ObjectCreated*, *ObjectDestroyed*, *OnIceCandidate*, *OnIceComponentStateChanged*, *OnIceGatheringDone*)
    - *object* (required, object).Media element identifier.
    - *type* (required, string). Type identifier (same value than before)

The following example shows a notification sent for server to client to notify an event of type *EndOfStream* in a *PlayerEndpoint* object:

- Body (application/json)

```
{
    "jsonrpc": "2.0",
    "method": "onEvent",
    "params": {
        "value": {
            "data": {
                "source": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline/
↪76dcb8d7-5655-445b-8cb7-cf5dc91643bc_kurento.PlayerEndpoint",
                "tags": [],
                "timestamp": "1461589478",
                "type": "EndOfStream"
```

(continues on next page)

(continued from previous page)

```
            },
            "object": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline/
↪76dcb8d7-5655-445b-8cb7-cf5dc91643bc_kurento.PlayerEndpoint",
            "type": "EndOfStream"
        }
    }
}
```

Notice that this message has no *id* field due to the fact that no response is required.

1.4.7.2 Response

There is no response to the *onEvent* message.

1.5 Glossary

This is a glossary of terms that often appear in discussion about multimedia transmissions. Some of the terms are specific to *GStreamer* or *Kurento*, and most of them are described and linked to their RFC, W3C or Wikipedia documents.

Agnostic media One of the big problems of media is that the number of variants of video and audio codecs, formats and variants quickly creates high complexity in heterogeneous applications. So Kurento developed the concept of an automatic converter of media formats that enables development of *agnostic* elements. Whenever a media element's source is connected to another media element's sink, the Kurento framework verifies if media adaption and transcoding is necessary and, if needed, it transparently incorporates the appropriate transformations making possible the chaining of the two elements into the resulting *Pipeline*.

AVI Audio Video Interleaved, known by its initials AVI, is a multimedia container format introduced by Microsoft in November 1992 as part of its Video for Windows technology. AVI files can contain both audio and video data in a file container that allows synchronous audio-with-video playback. AVI is a derivative of the Resource Interchange File Format (RIFF).

See also:

Bower Bower is a package manager for the web. It offers a generic solution to the problem of front-end package management, while exposing the package dependency model via an API that can be consumed by a build stack.

Builder Pattern The builder pattern is an object creation software design pattern whose intention is to find a solution to the telescoping constructor anti-pattern. The telescoping constructor anti-pattern occurs when the increase of object constructor parameter combination leads to an exponential list of constructors. Instead of using numerous constructors, the builder pattern uses another object, a builder, that receives each initialization parameter step by step and then returns the resulting constructed object at once.

See also:

CORS Cross-origin resource sharing is a mechanism that allows JavaScript code on a web page to make XMLHttpRequests to different domains than the one the JavaScript originated from. It works by adding new HTTP headers that allow servers to serve resources to permitted origin domains. Browsers support these headers and enforce the restrictions they establish.

See also:

enable-cors.org Information on the relevance of CORS and how and when to enable it.

DOM Document Object Model is a cross-platform and language-independent convention for representing and interacting with objects in HTML, XHTML and XML documents.

EOS End Of Stream is an event that occurs when playback of some media source has finished. In Kurento, some elements will raise an `EndOfStream` event.

GStreamer GStreamer is a pipeline-based multimedia framework written in the C programming language.

H.264 A Video Compression Format. The H.264 standard can be viewed as a "family of standards" composed of a number of profiles. Each specific decoder deals with at least one such profiles, but not necessarily all.

See also:

RFC 6184 RTP Payload Format for H.264 Video (This RFC obsoletes **RFC 3984**)

HTTP The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web.

See also:

RFC 2616 Hypertext Transfer Protocol – HTTP/1.1

ICE Interactive Connectivity Establishment (ICE) is a technique used to achieve *NAT Traversal*. ICE makes use of the *STUN* protocol and its extension, *TURN*. ICE can be used by any aplication that makes use of the SDP Offer/Answer model..

See also:

RFC 5245 Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols

IMS IP Multimedia Subsystem (IMS) is the 3GPP's Mobile Architectural Framework for delivering IP Multimedia Services in 3G (and beyond) Mobile Networks.

See also:

RFC 3574 Transition Scenarios for 3GPP Networks

jQuery jQuery is a cross-platform JavaScript library designed to simplify the client-side scripting of HTML.

JSON JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is designed to be easy to understand and write for humans and easy to parse for machines.

JSON-RPC JSON-RPC is a simple remote procedure call protocol encoded in JSON. JSON-RPC allows for notifications and for multiple calls to be sent to the server which may be answered out of order.

Kurento Kurento is a platform for the development of multimedia-enabled applications. Kurento is the Esperanto term for the English word 'stream'. We chose this name because we believe the Esperanto principles are inspiring for what the multimedia community needs: simplicity, openness and universality. Some components of Kurento are the *Kurento Media Server*, the *Kurento API*, the *Kurento Protocol*, and the *Kurento Client*.

Kurento API An object oriented API to create media pipelines to control media. It can be seen as and interface to Kurento Media Server. It can be used from the Kurento Protocol or from Kurento Clients.

Kurento Client A programming library (Java or JavaScript) used to control an instance of **Kurento Media Server** from an application. For example, with this library, any developer can create a web application that uses Kurento Media Server to receive audio and video from the user web browser, process it and send it back again over Internet. The Kurento Client libraries expose the *Kurento API* to application developers.

Kurento Protocol Communication between KMS and clients by means of *JSON-RPC* messages. It is based on *WebSocket* that uses *JSON-RPC* v2.0 messages for making requests and sending responses.

KMS

Kurento Media Server **Kurento Media Server** is the core element of Kurento since it responsible for media transmission, processing, loading and recording.

Maven Maven is a build automation tool used primarily for Java projects.

Media Element A **Media Element** is a module that encapsulates a specific media capability. For example **RecorderEndpoint**, **PlayerEndpoint**, etc.

Media Pipeline A Media Pipeline is a chain of media elements, where the output stream generated by one element (source) is fed into one or more other elements input streams (sinks). Hence, the pipeline represents a "machine" capable of performing a sequence of operations over a stream.

Media Plane In a traditional IP Multimedia Subsystem, the handling of media is conceptually splitted in two layers. The layer that handles the media itself -with functionalities such as media transport, encoding/decoding, and processing- is called Media Plane.

See also:

Signaling Plane

MP4 MPEG-4 Part 14 or MP4 is a digital multimedia format most commonly used to store video and audio, but can also be used to store other data such as subtitles and still images.

See also:

Multimedia Multimedia is concerned with the computer controlled integration of text, graphics, video, animation, audio, and any other media where information can be represented, stored, transmitted and processed digitally. There is a temporal relationship between many forms of media, for instance audio, video and animations. There 2 are forms of problems involved in

- Sequencing within the media, i.e. playing frames in correct order or time frame.
- Synchronization, i.e. inter-media scheduling. For example, keeping video and audio synchronized or displaying captions or subtitles in the required intervals.

See also:

Multimedia container format Container or wrapper formats are meta-file formats whose specification describes how different data elements and metadata coexist in a computer file. Simpler multimedia container formats can contain different types of audio formats, while more advanced container formats can support multiple audio and video streams, subtitles, chapter-information, and meta-data, along with the synchronization information needed to play back the various streams together. In most cases, the file header, most of the metadata and the synchro chunks are specified by the container format.

See also:

NAT

Network Address Translation Network address translation (NAT) is the technique of modifying network address information in Internet Protocol (IP) datagram packet headers while they are in transit across a traffic routing device for the purpose of remapping one IP address space into another.

See also:

NAT-T

NAT Traversal NAT traversal (sometimes abbreviated as NAT-T) is a general term for techniques that establish and maintain Internet protocol connections traversing network address translation (NAT) gateways, which break end-to-end connectivity. Intercepting and modifying traffic can only be performed transparently in the absence of secure encryption and authentication.

See also:

NAT Types and NAT Traversal Entry in Kurento Knowledge Base.

NAT Traversal White Paper White paper on NAT-T and solutions for end-to-end connectivity in its presence

Node.js Node.js is a cross-platform runtime environment for server-side and networking applications. Node.js applications are written in JavaScript, and can be run within the Node.js runtime on OS X, Microsoft Windows and Linux with no changes.

npm npm is the official package manager for *Node.js*.

OpenCL OpenCL is the standard framework for cross-platform, parallel programming of heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors.

OpenCV OpenCV (Open Source Computer Vision Library) is a BSD-licensed open source computer vision and machine learning software library. OpenCV aims to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception.

Pad, Media A Media Pad is is an element's interface with the outside world. Data streams from the MediaSource pad to another element's MediaSink pad.

See also:

GStreamer Pad Definition of the Pad structure in GStreamer

PubNub PubNub is a publish/subscribe cloud service for sending and routing data. It streams data to global audiences on any device using persistent socket connections. PubNub has been designed to deliver data with low latencies to end-user devices. These devices can be behind firewalls, NAT environments, and other hard-to-reach network environments. PubNub provides message caching for retransmission of lost signals over unreliable network environments. This is accomplished by maintaining an always open socket connection to every device.

QR QR code (Quick Response Code) is a type of two-dimensional barcode. that became popular in the mobile phone industry due to its fast readability and greater storage capacity compared to standard UPC barcodes.

See also:

REMB **Receiver Estimated Maximum Bitrate** (REMB) is a type of RTCP feedback message that a RTP receiver can use to inform the sender about what is the estimated reception bandwidth currently available for the stream itself. Upon reception of this message, the RTP sender will be able to adjust its own video bitrate to the conditions of the network. This message is a crucial part of the *Google Congestion Control* (GCC) algorithm, which provides any RTP session with the ability to adapt in cases of network congestion.

The *GCC* algorithm is one of several proposed algorithms that have been proposed by an IETF Working Group named *RTP Media Congestion Avoidance Techniques* (RMCAT).

See also:

What is RMCAT congestion control, and how will it affect WebRTC?

draft-alvestrand-rmcat-remb RTCP message for Receiver Estimated Maximum Bitrate

draft-ietf-rmcat-gcc A Google Congestion Control Algorithm for Real-Time Communication

REST Representational state transfer (REST) is an architectural style consisting of a coordinated set of constraints applied to components, connectors, and data elements, within a distributed hypermedia system. The term representational state transfer was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation.

See also:

RTCP The RTP Control Protocol (RTCP) is a sister protocol of the *RTP*, that provides out-of-band statistics and control information for an RTP flow.

See also:

RFC 3605 Real Time Control Protocol (RTCP) attribute in Session Description Protocol (SDP)

RTP Real-time Transport Protocol (RTP) is a standard packet format designed for transmitting audio and video streams on IP networks. It is used in conjunction with the *RTP Control Protocol*. Transmissions using the RTP audio/video profile (RTP/AVP) typically use *SDP* to describe the technical parameters of the media streams.

See also:

RFC 3550 RTP: A Transport Protocol for Real-Time Applications

Same-origin policy The "same-origin policy" is a web application security model. The policy permits scripts running on pages originating from the same domain to access each other's *DOM* with no specific restrictions, but prevents access to *DOM* on different domains.

See also:

SDP

Session Description Protocol

SDP Offer/Answer The **Session Description Protocol** (SDP) is a text document that describes the parameters of a streaming media session. It is commonly used to describe the characteristics of RTP streams (and related protocols such as RTSP).

The **SDP Offer/Answer** model is a negotiation between two peers of a unicast stream, by which the sender and the receiver share the set of media streams and codecs they wish to use, along with the IP addresses and ports they would like to use to receive the media.

This is an example SDP Offer/Answer negotiation. First, there must be a peer that wishes to initiate the negotiation; we'll call it the *offerer*. It composes the following SDP Offer and sends it to the other peer -which we'll call the *answerer*-:

```
v=0
o=- 0 0 IN IP4 127.0.0.1
s=Example sender
c=IN IP4 127.0.0.1
t=0 0
m=audio 5006 RTP/AVP 96
a=rtpmap:96 opus/48000/2
a=sendonly
m=video 5004 RTP/AVP 103
a=rtpmap:103 H264/90000
a=sendonly
```

Upon receiving that Offer, the *answerer* studies the parameters requested by the *offerer*, decides if they can be satisfied, and composes an appropriate SDP Answer that is sent back to the *offerer*:

```
v=0
o=- 3696336115 3696336115 IN IP4 192.168.56.1
s=Example receiver
c=IN IP4 192.168.56.1
t=0 0
m=audio 0 RTP/AVP 96
a=rtpmap:96 opus/48000/2
a=recvonly
m=video 31278 RTP/AVP 103
a=rtpmap:103 H264/90000
a=recvonly
```

The SDP Answer is the final step of the SDP Offer/Answer Model. With it, the *answerer* agrees to some of the parameter requested by the *offerer*, but not all.

In this example, `audio 0` means that the *answerer* rejects the audio stream that the *offerer* intended to send; also, it accepts the video stream, and the `a=recvonly` acknowledges that the *answerer* will exclusively act as a receiver, and won't send any stream back to the other peer.

See also:

Anatomy of a WebRTC SDP

RFC 4566 SDP: Session Description Protocol

RFC 4568 Session Description Protocol (SDP) Security Descriptions for Media Streams

Semantic Versioning Semantic Versioning is a formal convention for specifying compatibility using a three-part version number: major version; minor version; and patch.

Signaling Plane It is the layer of a media system in charge of the information exchanges concerning the establishment and control of the different media circuits and the management of the network, in contrast to the transfer of media, done by the Signaling Plane. Functions such as media negotiation, QoS parametrization, call establishment, user registration, user presence, etc. as managed in this plane.

See also:

Media Plane

Sink, Media A Media Sink is a MediaPad that outputs a Media Stream. Data streams from a MediaSource pad to another element's MediaSink pad.

SIP Session Initiation Protocol (SIP) is a *signaling plane* protocol widely used for controlling multimedia communication sessions such as voice and video calls over Internet Protocol (IP) networks. SIP works in conjunction with several other application layer protocols:

- *SDP* for media identification and negotiation.
- *RTP*, *SRTP* or *WebRTC* for the transmission of media streams.
- A *TLS* layer may be used for secure transmission of SIP messages.

See also:

Source, Media A Media Source is a Media Pad that generates a Media Stream.

SPA

Single-Page Application A single-page application is a web application that fits on a single web page with the goal of providing a more fluid user experience akin to a desktop application.

Sphinx Sphinx is a documentation generation system. Text is first written using reStructuredText markup language, which then is transformed by Sphinx into different formats such as PDF or HTML. This is the documentation tool of choice for the Kurento project.

See also:

Easy and beautiful documentation with Sphinx

Spring Boot Spring Boot is Spring's convention-over-configuration solution for creating stand-alone, production-grade Spring based applications that can you can "just run". It embeds Tomcat or Jetty directly and so there is no need to deploy WAR files in order to run web applications.

SRTCP SRTCP provides the same security-related features to RTCP, as the ones provided by SRTP to RTP. Encryption, message authentication and integrity, and replay protection are the features added by SRTCP to *RTCP*.

See also:

SRTP

SRTP Secure RTP is a profile of RTP (*Real-time Transport Protocol*), intended to provide encryption, message authentication and integrity, and replay protection to the RTP data in both unicast and multicast applications. Similarly to how RTP has a sister RTCP protocol, SRTP also has a sister protocol, called Secure RTCP (or *SRTCP*).

See also:

RFC 3711 The Secure Real-time Transport Protocol (SRTP)

SSL Secure Socket Layer. See *TLS*.

STUN STUN stands for **Session Traversal Utilities for NAT**. It is a standard protocol (IETF RFC 5389) used by *NAT* traversal algorithms to assist hosts in the discovery of their public network information. If the routers between peers use full cone, address-restricted, or port-restricted NAT, then a direct link can be discovered with STUN alone. If either one of the routers use symmetric NAT, then a link can be discovered with STUN packets only if the other router does not use symmetric or port-restricted NAT. In this later case, the only alternative left is to discover a relayed path through the use of *TURN*.

Trickle ICE Extension to the *ICE* protocol that allows ICE agents to send and receive candidates incrementally rather than exchanging complete lists. With such incremental provisioning, ICE agents can begin connectivity checks while they are still gathering candidates and considerably shorten the time necessary for ICE processing to complete.

See also:

draft-ietf-ice-trickle Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol

TLS Transport Layer Security (TLS) and its predecessor Secure Socket Layer (SSL).

See also:

RFC 5246 The Transport Layer Security (TLS) Protocol Version 1.2

TURN TURN stands for **Traversal Using Relays around NAT**. Like *STUN*, it is a network protocol (IETF RFC 5766) used to assist in the discovery of paths between peers on the Internet. It differs from STUN in that it uses a public intermediary relay to act as a proxy for packets between peers. It is used when no other option is available since it consumes server resources and has an increased latency. The only time when TURN is necessary is when one of the peers is behind a symmetric NAT and the other peer is behind either a symmetric NAT or a port-restricted NAT.

VP8 VP8 is a video compression format created by On2 Technologies as a successor to VP7. Its patents rights are owned by Google, who made an irrevocable patent promise on its patents for implementing it and released a specification under the Creative Commons Attribution 3.0 license.

See also:

RFC 6386 VP8 Data Format and Decoding Guide

WebM WebM is an open media file format designed for the web. WebM files consist of video streams compressed with the VP8 video codec and audio streams compressed with the Vorbis audio codec. The WebM file structure is based on the Matroska media container.

WebRTC WebRTC is a set of protocols, mechanisms and APIs that provide browsers and mobile applications with Real-Time Communications (RTC) capabilities over peer-to-peer connections.

See also:

WebRTC Working Draft

WebSocket WebSocket specification (developed as part of the HTML5 initiative) defines a full-duplex single socket connection over which messages can be sent between client and server.

Index

S

T

V

W