FIWARE-Stream-Oriented-GE

Release

Contents

Introduction

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.

Table of Contents

2.1 FIWARE Stream Oriented Generic Enabler - Programmers Guide

2.1.1 Introduction

The *Stream Oriented GE Kurento* is a multimedia platform aimed to help developers to add multimedia capabilities to their applications. The core element is the *Kurento Media Server* (KMS), a Gstreamer based multimedia engine that provides the following features:

- Networked streaming protocols, including *HTTP*, *RTP* and *WebRTC*.

- Media transcodification between any of the codecs currently supported by Gstreamer.

- Generic support for computational vision and augmented reality filters.

- Media storage supporting writing operations for *WebM* and *MP4* and reading operations for any of *Gstreamer's* muxers.

Kurento *Java* and *JavaScript* clients are available for developers. These clients are libraries to connect to KMS and this way incorporate the above features in applications.

Background and Detail

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.

2.1.2 Programmer Guide

The *Stream Oriented GE Kurento* software is released under LGPL version 2.1 license. This is quite a convenient license for programmers, but it is still recommended you check if it actually fits your application needs.

Basic Setup

First of all, developers must install Kurento Media Server. Please review the installation guide. In short, KMS 6.6.0 can be installed and started in an Ubuntu 14.04 machine as follows:

```
echo "deb http://ubuntu.kurento.org trusty kms6" | sudo tee /etc/apt/sources.list.d/kurento.list
wget -O - http://ubuntu.kurento.org/kurento.gpg.key | sudo apt-key add -
sudo apt-get update
```

```
sudo apt-get install kurento-media-server-6.0
sudo service kurento-media-server-6.0 start
```

Once a Kurento Media Server is installed, you need a *Kurento Client* to create your own applications with advanced media capabilities. A Kurento Client is a programming library used to control the Kurento Media Server from an application. Communication between a Kurento Client and the Kurento Media Server is implemented by the Stream Oriented GE Open API. This communication beetween Kurento Clients and Kurento Media Server is done by means of a *WebSocket*.

There are available Kurento Client libraries in *Java* and *JavaScript*. These libraries are based on the concept of *Media Element*. A Media Element holds a specific media capability. For example, the media element called *WebRtcEndpoint* holds the capability of sending and receiving WebRTC media streams, the media element called *RecorderEndpoint* has the capability of recording into the file system any media streams it receives, the *FaceOverlayFilter* detects faces on the exchanged video streams and adds a specific overlaid image on top of them, etc. Kurento exposes a rich toolbox of media elements as part of its APIs.

The following sections provides information for create web applications with the Stream Oriented GE using Java and JavaScript.

Programming with the Stream Oriented GE in Java

The Kurento Java Client is provided as Maven dependency in Maven Central repository. To use it in a Maven application you have to include the following dependencies in your *pom.xml*:

```
<dependencies>
    <dependency>
        <groupId>org.kurento</groupId>
        <artifactId>kurento-client</artifactId>
        <version>6.6.0</version>
    </dependency>
</dependencies>
```

KurentoClient is the Java class used to connect to Kurento Media Server. This class has several static factory methods to create instances from it. In the following code snippet you can see how to create a *KurentoClient* instance:

```
KurentoClient kurento = KurentoClient.create("ws://localhost:8888/kurento");
```

A *MediaPipeline* object is required to build media services. The method *createMediaPipeline()* of a *KurentoClient* object can be used for this purpose:

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

Media elements within a pipeline can be connected to build services, but they are isolated from the rest of the system. Media elements are created using the builder pattern allowing a flexible initialization. Mandatory parameters must be provided in the builder constructor. Optional parameters are set to defaults unless the application overrides their values using setter methods in the builder object. When the builder is configured, the object can be created using its *build()* method. In the following snippet, several media elements are created:

```
// Protocols and codecs
WebRtcEndpoint webRtcEndpoint = new WebRtcEndpoint.Builder(pipeline).build();


HttpPostEndpoint httpPostEndpoint = new HttpPostEndpoint.Builder(pipeline).build();


RtpEndpoint rtpEndpoint = new RtpEndpoint.Builder(pipeline).build();


// Media repository
PlayerEndpoint playerEndpoint = new PlayerEndpoint.Builder(pipeline, "http://files.kurento.org/video/
```

```
RecorderEndpoint recorderEndpoint = new RecorderEndpoint.Builder(pipeline, "file:///tmp/recording.wek

// Filters
FaceOverlayFilter faceOverlayFilter = new FaceOverlayFilter.Builder(pipeline).build();

ZBarFilter zBarFilter = new ZBarFilter.Builder(pipeline).build();

GStreamerFilter gstreamerFilter = new GStreamerFilter.Builder(pipeline, "videoflip method=4").build()

// Group communications
Composite composite = new Composite.Builder(pipeline).build();

Dispatcher dispatcher = new Dispatcher.Builder(pipeline).build();

DispatcherOneToMany dispatcherOneToMany = new DispatcherOneToMany.Builder(pipeline).build();
```

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. Hence, when creating a pipeline, developers need to determine the capabilities they want to use (the media elements) and the topology determining which media elements provide media to which other media elements (the connectivity). The connectivity is controlled through the *connect* primitive, exposed on all Kurento Client APIs. This primitive is always invoked in the element acting as *source* and takes as argument the *sink* element following this scheme:

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

Programming with the Stream Oriented GE in JavaScript

The Kurento JavaScript Client is provided as Bower dependency in Bower repository. To use it in a Bower application you have to include the following dependencies in your *bower.json*:

```
"dependencies": {
   "kurento-client": "6.6.0",
}
```

First, you need to create an instance of the *KurentoClient* class that will manage the connection with the Kurento Media Server, so you need to provide the URI of its WebSocket:

```
kurentoClient(ws_uri, function(error, kurentoClient) {
   if (error) {
      // Error connecting to KMS
   }

   // Success connecting to KMS

});
```

The second step is to create a pipeline using the previously created *kurentoClient*, as follows:

```
kurentoClient.create('MediaPipeline', function(error, pipeline) {
   if (error) {
      // Error creating MediaPipeline
   }

   // Success creating MediaPipeline
});
```

Then we should create the media elements. The following snippet shows how to create several media elements:

```
// Protocols and codecs
pipeline.create('WebRtcEndpoint', function(error, webRtcEndpoint) {
   if (error) {
      // Error creating WebRtcEndpoint
   }

   // Success creating WebRtcEndpoint
});

pipeline.create('HttpPostEndpoint', function(error, httpPostEndpoint) {
   if (error) {
      // Error creating HttpPostEndpoint
   }

   // Success creating HttpPostEndpoint
});

pipeline.create('RtpEndpoint', function(error, rtpEndpoint) {
   if (error) {
      // Error creating RtpEndpoint
   }

   // Success creating RtpEndpoint
});

// Media repository
pipeline.create('PlayerEndpoint', {uri : 'http://files.kurento.org/video/filter/fiwarecut.mp4'}, func
   if (error) {
      // Error creating PlayerEndpoint
   }

   // Success creating PlayerEndpoint
});

pipeline.create('RecorderEndpoint', {uri : 'file:///tmp/recording.webm'}, function(error, recorderEnd
   if (error) {
      // Error creating RecorderEndpoint
   }

  // Success creating RecorderEndpoint
});

// Filters
pipeline.create('FaceOverlayFilter', function(error, faceOverlayFilter) {
   if (error) {
      // Error creating FaceOverlayFilter
   }

   // Success creating FaceOverlayFilter
});

pipeline.create('ZBarFilter', function(error, zBarFilter) {
   if (error) {
      // Error creating ZBarFilter
   }

  // Success creating WebRtcEndpoint
});
```

```
pipeline.create('GStreamerFilter', {command : 'videoflip method=4'}, function(error, recorderEndpoint
   if (error) {
      // Error creating GStreamerFilter
   }

   // Success creating GStreamerFilter
});

// Group communications
pipeline.create('Composite', function(error, composite) {
   if (error) {
      // Error creating Composite
   }

  // Success creating Composite
});

pipeline.create('Dispatcher', function(error, dispatcher) {
   if (error) {
      // Error creating Dispatcher
   }

   // Success creating Dispatcher
});

pipeline.create('DispatcherOneToMany', function(error, dispatcherOneToMany) {
   if (error) {
      // Error creating DispatcherOneToMany
   }

   // Success creating DispatcherOneToMany
});
```

Finally, media elements can be connected. The method *connect()* of the Media Elements is always invoked in the element acting as *source* and takes as argument the as *sink* element. For example a *WebRtcEndpoint* connected to itself (loopback):

```
webRtc.connect(webRtc, function(error) {
   if (error) {
      // Error connecting media elements
   }

   // Success connecting media elements
});
```

Magic-Mirror Example

The *Magic-Mirror* web application is a good example to introduce the principles of programming with Kurento. This application uses computer vision and augmented reality techniques to add a funny hat on top of faces.The following picture shows a screenshot of the demo running in a web browser:

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

The logic of the application is quite simple: the local stream is sent to the Kurento Media Server, which returns it back to the client with a filter processing. This filtering consists in faces detection and overlaying of an image on the

top of them. To implement this behavior we need to create a Media Pipeline composed by two Media Elements: a *WebRtcEndpoint* connected to an *FaceOverlayFilter*. This filter element is connected again to the *WebRtcEndpoint*'s *sink* and then the stream is send back (to browser). This media pipeline is illustrated in the following picture:
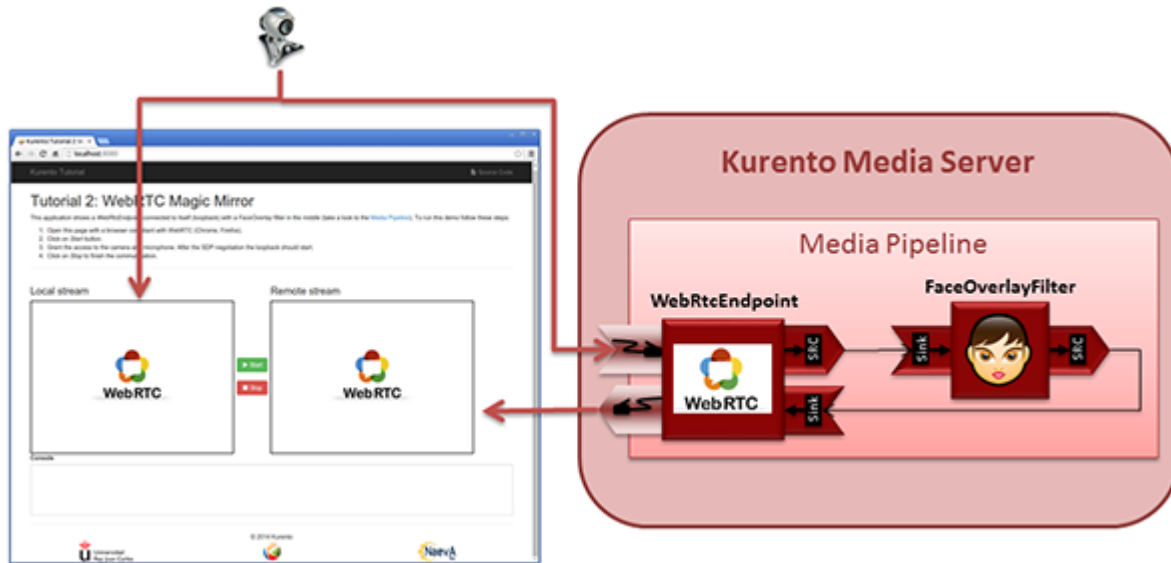
This demo has been implemented in Java, Javascript, and also Node.js. *Java* implementation uses the *Kurento Java Client*, while *JavScript* and *Node.js* uses the *Kurento JavaScript Client*. In addition, these three demos use *Kurento JavaScript Utils* library in the client-side. This is an utility JavaScript library aimed to simplify the development of WebRTC applications. In these demos, the function *WebRtcPeer.startSendRecv* is used to abstract the WebRTC internal details (i.e. *PeerConnection* and *getUserStream*) and makes possible to start a full-duplex WebRTC communication.

The *Java* version is hosted on GitHub. To run this demo in an Ubuntu machine, execute the following commands in the shell:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git
cd kurento-tutorial-java/kurento-magic-mirror
git checkout 6.6.0
mvn compile exec:java
```

The pre-requisites to run this Java demo are Git, JDK 7, Maven, and Bower. To install these tools in Ubuntu please execute these commands:

```
sudo apt-get install git
sudo apt-get install openjdk-7-jdk
sudo apt-get install maven
curl -sL https://deb.nodesource.com/setup | sudo bash -
sudo apt-get install -y nodejs
sudo npm install -g bower
```

The *JavaScript* version is hosted on GitHub. To run this demo in an Ubuntu machine, execute the following commands in the shell:

```
git clone https://github.com/Kurento/kurento-tutorial-js.git
cd kurento-tutorial-js/kurento-magic-mirror
git checkout 6.6.0
bower install
http-server
```

The pre-requisites to run this JavaScript demo are Git, Node.js, Bower, and a HTTP Server, for example a Node.js http-servert:

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

The *Node.js* version is hosted on GitHub. To run this demo in an Ubuntu machine, execute the following commands in the shell:

```
git clone https://github.com/Kurento/kurento-tutorial-node.git
cd kurento-tutorial-node/kurento-magic-mirror
git checkout 6.6.0
npm install
npm start
```

The pre-requisites to run this Node.js demo are Git, Node.js, and Bower:

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

Finally, open the demo (Java, JavaScript or Node.js) in the URL https://localhost:8443/ with a capable WebRTC browser, for example, Google Chrome. To install it in Ubuntu (64 bits):

```
sudo apt-get install libxss1
wget https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb
sudo dpkg -i google-chrome*.deb
```

More Examples

There are another sample applications that can be used to learn how to use the *Stream Oriented GE Kurento*, namely:

- Hello-world application. This is one of the simplest WebRTC application you can create with Kurento. It implements a WebRTC loopback (a WebRTC media stream going from client to Kurento and back to the client). You can check out the source code on GitHub for Java, Browser JavaScript and Node.js.

- One to many video call application. This web application consists video broadcasting with WebRTC. One peer transmits a video stream and N peers receives it. This web application is a videophone (call one to one) based on WebRTC. You can check out the source code on GitHub for Java and Node.js.

- One to one video call. You can check out the source code on GitHub for Java and Node.js.

- Advanced one to one video call application. This is an enhanced version of the previous application recording of the video communication, and also integration with an augmented reality filter. You can check out the source code on GitHub for Java.

2.2 FIWARE Stream Oriented Generic Enabler - Installation and Administration Guide

2.2.1 Introduction

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.

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)

Operating System

Kurento Media Server has to be installed on Ubuntu 14.04 LTS (64 bits).

Dependencies

If end-to-end testing is going to be performed, the following tools must be also installed in your system (Ubuntu):

- Open JDK 7:

```
sudo apt-get update
sudo apt-get install openjdk-7-jdk
```

- Git:

```
sudo apt-get install git
```

- Chrome (latest stable version):

```
sudo apt-get install libxss1
wget https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb
sudo dpkg -i google-chrome*.deb
```

If you encounter any errors with the commands above, simply use:

```
sudo apt-get -f install
```

- Maven:

```
sudo apt-get install maven
```

- Bower:

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

2.2.2 Installation

In order to install and start the latest stable Kurento Media Server version (6.6.0) you have to type the following commands, one at a time and in the same order as listed here. When asked for any kind of confirmation, reply affirmatively:

```
echo "deb http://ubuntu.kurento.org trusty kms6" | sudo tee /etc/apt/sources.list.d/kurento.list
wget -O - http://ubuntu.kurento.org/kurento.gpg.key | sudo apt-key add -
sudo apt-get update
sudo apt-get install kurento-media-server-6.0
sudo service kurento-media-server-6.0 start
```

After running these commands, Kurento Media Server should be installed and started.

2.2.3 Configuration

The main Kurento Media Server configuration file is located in */etc/kurento/kurento.conf.json*. After a fresh installation this file is the following:

```
{
  "mediaServer" : {
    "resources": {
    //  //Resources usage limit for raising an exception when an object creation is attempted
    //  "exceptionLimit": "0.8",
    //  // Resources usage limit for restarting the server when no objects are alive
    //  "killLimit": "0.7",
        // Garbage collector period in seconds
        "garbageCollectorPeriod": 240
    },
    "net" : {
      "websocket": {
        "port": 8888,
        //"secure": {
        //  "port": 8433,
        //  "certificate": "defaultCertificate.pem",
        //  "password": ""
        //},
        //"registrar": {
        //  "address": "ws://localhost:9090",
        //  "localAddress": "localhost"
        //},
        "path": "kurento",
        "threads": 10
      }
    }
  }
}
```

As of Kurento Media Server version 6, in addition to this general configuration file, the specific features of KMS are tuned as individual modules. Each of these modules has its own configuration file:

- */etc/kurento/modules/kurento/MediaElement.conf.ini*: Generic parameters for Media Elements.

- */etc/kurento/modules/kurento/SdpEndpoint.conf.ini*: Audio/video parameters for SdpEndpoints (i.e. *WebRtcEndpoint* and *RtpEndpoint*).

- */etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini*: Specific parameters for *WebRtcEndpoint*.

- */etc/kurento/modules/kurento/HttpEndpoint.conf.ini*: Specific parameters for *HttpEndpoint*.

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.

In order to setup a STUN server you should uncomment the following lines in the Kurento Media Server configuration file located on at */etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini*:

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

In order to setup a TURN server you should uncomment the following lines in the Kurento Media Server configuration file located on at */etc/kurento/modules/kurento/WebRtcEndpoint.conf.ini*:

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

As before, TURN address should be an IP address (not domain name). See some examples of TURN configuration below:

```
turnURL=kurento:kurento@193.147.51.36:3478
```

... or using a free access numb STUN/TURN server as follows:

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

An open source implementation of a TURN server is coturn.

2.2.4 Sanity check Procedures

End to End testing

Kurento Media Server must be installed and started before running the following example, which is called *magic-mirror* and it is developed with the *Kurento Java Client*. You should run this example in a machine with camera and

---

microphone since live media is needed. To launch the application first you need to clone the GitHub project where it is hosted and then run the main class, as follows:

```
git clone https://github.com/Kurento/kurento-tutorial-java.git
cd kurento-tutorial-java/kurento-magic-mirror
git checkout 6.6.0
mvn compile exec:java
```

Note: In order to run this example, be sure that you have installed the dependencies (Kurento Media Server, JDK, Git, Chrome, Maven, and Bower) as described in the section before.

These commands starts an HTTP server at the localhost in the port 8443. Therefore, please open the web application connecting to the URL https://localhost:8443/ through a WebRTC capable browser (e.g. Chrome). Click on the *Start* button and grant the access to the camera and microphone. After the SDP negotiation an enhanced video mirror should start. Kurento Media Server is processing media in real time, detecting faces and overlying an image on the top of them. This is a simple example of augmented reality in real time with Kurento.

Take into account that this setup is assuming that port TCP 8443 is available in your system. If you would like to use another one, simply launch the demo as follows:

```
mvn compile exec:java -Dserver.port=<custom-port>
```

... and open the application on http://localhost:custom-port/.

List of Running Processes

To verify that Kurento Media Server is up and running use the command:

```
ps -ef | grep kurento
```

The output should include the kurento-media-server process:

```
kurento    1270    1  0 08:52 ?        00:01:00 /usr/bin/kurento-media-server
```

Network interfaces Up & Open

Unless configured otherwise, Kureno Media Server will open the port TCP 8888 to receive requests and send responses to/from by means of the Kurento clients (by means of the Kurento Protocol Open API). To verify if this port is listening, execute the following command:

```
sudo netstat -putan | grep kurento
```

The output should be similar to the following:

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

2.2.5 Diagnosis Procedures

Resource consumption

Resource consumption documented in this section has been measured in two different scenarios:

- Low load: all services running, but no stream being served.

- High load: heavy load scenario where 20 streams are requested at the same time.

Under the above circumstances, the *top* command showed the following results in the hardware described below:

Machine Type	Physical Machine
CPU	Intel(R) Core(TM) i5-3337U CPU @ 1.80GHz
RAM	16 GB
HDD	500 GB
Operating System	Ubuntu 14.04

Kurento Media Server gave the following result:

	Low Usage	**Heavy Usage**
CPU	0.0 %	76.9 %
RAM	81.92 MB	655.36 MB

I/O flows

Use the following commands to start and stop Kurento Media Server respectively:

```
sudo service kurento-media-server-6.0 start
sudo service kurento-media-server-6.0 stop
```

Kurento Media Server logs file are stored in the folder */var/log/kurento-media-server/*. The content of this folder is as follows:

- *media-server_<timestamp>.<log_number>.<kms_pid>.log*: Current log for Kurento Media Server

- *media-server_error.log*: Third-party errors

- *logs*: Folder that contains the KMS rotated logs

When KMS starts correctly, this trace is written in the log file:

```
[time] [0x10b2f880] [info]    KurentoMediaServer main.cpp:255 main() Mediaserver started
```

2.3 FIWARE Stream Oriented Generic Enabler - Architecture Description

2.3.1 Copyright

Copyright © 2010-2016 by Unversidad Rey Juan Carlos. All Rights Reserved.

2.3.2 Legal Notice

Please check the following Legal Notice to understand the rights to use these specifications.

2.3.3 Overview

The Stream Oriented 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.

2.3.4 Main Concepts

Signaling and media planes

The Stream Oriented GE, as most multimedia communication technologies out there, is built upon two concepts that are key to all interactive communication systems:

- **Signaling Plane**. Module in charge of the management of communications, that is, it provides functions for media negotiation, QoS parametrization, call establishment, user registration, user presence, etc.

- **Media Plane**. Module in charge of the media itself. So, functionalities such as media transport, media encoding/decoding and media processing are part of it.

Media elements and media pipelines

The Stream Oriented GE is based on two concepts that act as building blocks for application developers:

- **Media Elements**:. A Media element is a functional unit performing a specific action on a media stream. Media elements are a way of every capability is represented as a self-contained "black box" (the media element) to the application developer, who does not need to understand the low-level details of the element for using it. Media elements are capable of receiving media from other elements (through media sources) and of sending media to other elements (through media sinks). Depending on their function, media elements can be split into different groups:

  - **Input Endpoints**: Media elements capable of receiving media and injecting it into a pipeline. There are several types of input endpoints. File input endpoints take the media from a file, Network input endpoints take the media from the network, and Capture input endpoints are capable of capturing the media stream directly from a camera or other kind of hardware resource.

  - **Filters**: Media elements in charge of transforming or analyzing media. Hence there are filters for performing operations such as mixing, muxing, analyzing, augmenting, etc.

  - **Hubs**: Media Objects in charge of managing multiple media flows in a pipeline. A Hub has several hub ports where other media elements are connected. Depending on the Hub type, there are different ways to

control the media. For example, there are a Hub called Composite that merge all input video streams in a unique output video stream with all inputs in a grid.

– **Output Endpoints**: Media elements capable of taking a media stream out of the pipeline. Again, there are several types of output endpoints specialized in files, network, screen, etc.

Fig. 2.1: *A media element is a functional unit providing a specific media capability, which is exposed to application developers as a "black box"*

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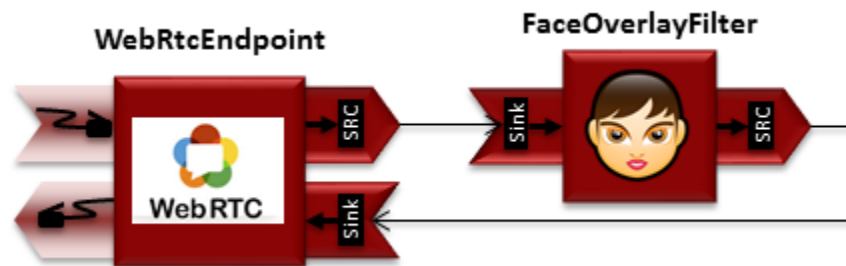

Fig. 2.2: *Example of a Media Pipeline implementing an interactive multimedia application receiving media from a WebRtcEndpoint, overlaying and image on the detected faces and sending back the resulting stream*

Agnostic media adapter

Using the Stream Oriented GE APIs, developers are able to compose the available media elements, getting the desired pipeline. There is a challenge in this scheme, as different media elements might require different input media formats than the output produced by their preceding element in the chain. For example, if we want to connect a WebRTC (VP8 encoded) or a RTP (H.264/H.263 encoded) video stream to a face recognition media element implemented to read raw RGB format, a transcoding is necessary.

Developers, specially during the initial phases of application development, might want to simplify development and abstract this heterogeneity, the Stream Oriented GE provides an automatic converter of media formats called the *agnostic media adapter*. Whenever a media element's source is connected to another media element's sink, our framework verifies if media adaption and transcoding is necessary and, in case it is, it transparently incorporates the appropriate transformations making possible the chaining of the two elements into the resulting pipeline.

Hence, this *agnostic media adapter* capability fully abstracts all the complexities of media codecs and formats. This may significantly accelerate the development process, specially when developers are not multimedia technology experts. However, there is a price to pay. Transcoding may be a very CPU expensive operation. The inappropriate design of pipelines that chain media elements in a way that unnecessarily alternate codecs (e.g. going from H.264, to raw, to H.264 to raw again) will lead to very poor performance of applications.
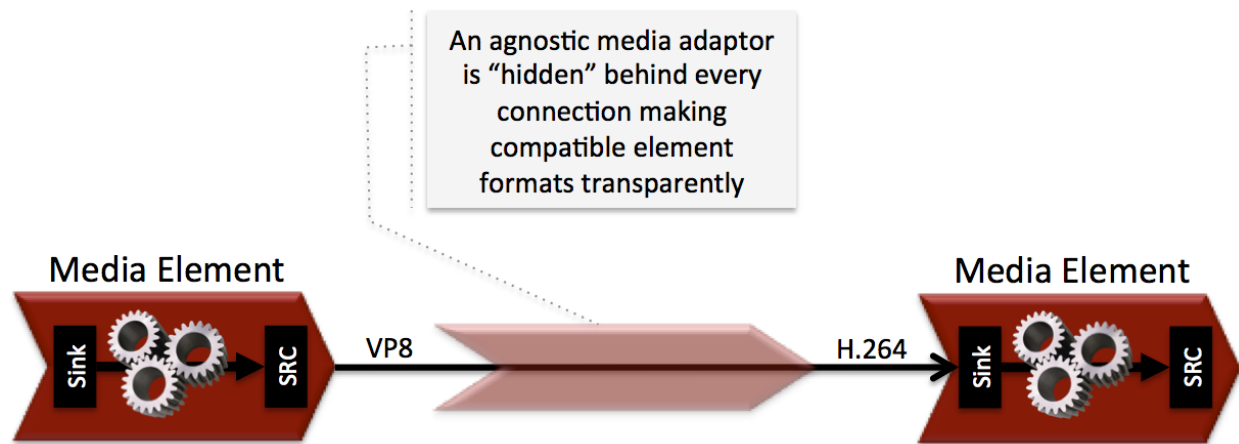
Fig. 2.3: *The agnostic media capability adapts formats between heterogeneous media elements making transparent for application developers all complexities of media representation and encoding.*

2.3.5 Generic Architecture

High level architecture

The conceptual representation of the GE architecture is shown in the following figure.

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 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 Media Server must be capable of managing the multimedia streams with minimal latency and maximum throughput. Hence the Media Server must be optimized for efficiency.

APIs and interfaces exposed by the architecture

The capabilities of the media plane (Media Server) and signaling plane (Application) are exposed through a number of APIs, which provide increasing abstraction levels. Following this, the role of the different APIs can be summarized in the following way:

- Stream Oriented GE Open API: Is a network protocol exposing the Media Server capabilities through Web-Socket (read more in the Stream Oriented Open API page).

- Java Client: Is a Java SE layer which consumes the Stream Oriented GE Open API and exposes its capabilities through a simple-to-use modularity based on Java POJOs representing media elements and media pipelines. This API is abstract in the sense that all the inherent complexities of the internal Open API workings are abstracted and developers do not need to deal with them when creating applications. Using the Java Client only requires adding the appropriate dependency to a maven project or to download the corresponding jar into the application developer CLASSPATH. It is important to remark that the Java Client is a media-plane control API. In other
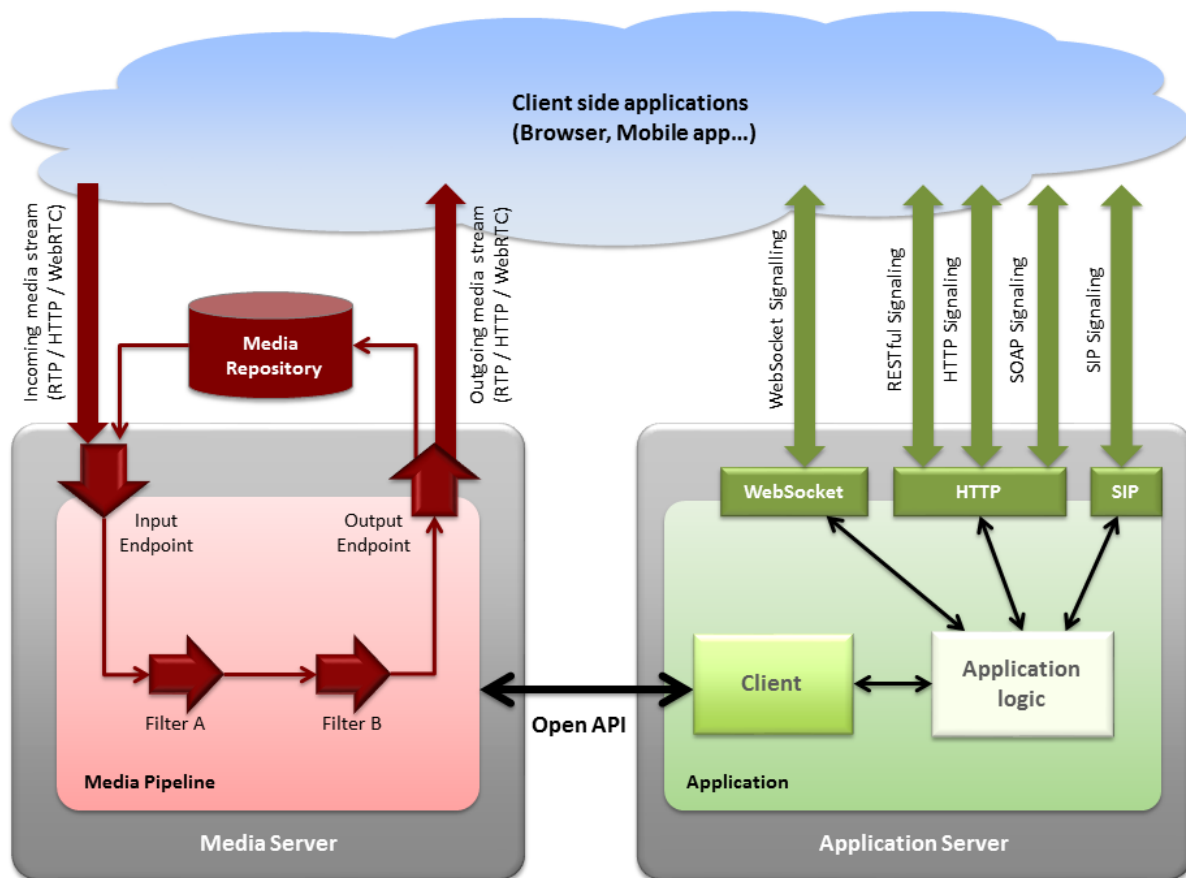
Fig. 2.4: *The Stream Oriented GE architecture follows the traditional separation between signaling and media planes.*

words, its objective is to expose the capability of managing media objects, but it does not provide any signaling plane capabilities.

- **JavaScript Client:** Is a JavaScript layer which consumes the Stream Oriented GE Open API and exposes its capabilities to JavaScript developers. It allow to build Node.js and browser based applications.

From an architectural perspective, application developers can use clients or the Open API directly for creating their multimedia enabled applications. This opens a wide spectrum of potential usage scenarios ranging from web applications (written using the JavaScript client), desktop applications (written using the Java Client), distributed applications (written using the Open API), etc.

2.3.6 Creating applications on top of the Stream Oriented GE Architecture

The Stream Oriented GE Architecture has been specifically designed following the architectural principles of the WWW. For this reason, creating a multimedia applications basing on it is 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**: 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 user input is captured. This usually consists on HTML pages.

- **Application logic**: This layer is in charge of implementing the specific functions executed by the application.

- **Service layer**: This layer provides capabilities used by the application logic such as databases, communications, security, etc.

Following this parallelism, multimedia applications created using the Stream Oriented GE also respond to the same architecture:

- **Presentation layer**: Is in charge of multimedia representation and multimedia capture. It is usually based on specific build-in capabilities of the client. For example, when creating a browser-based application, the presentation layer will use capabilities such as the <video> tag or the WebRTC PeerConnection and MediaStreams 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 Media Server (i.e. the specific media elements) is the part of the Stream Oriented GE architecture in charge of this layer.

This means that developers can choose to include the code creating the specific media pipeline required by their applications at the client side (using a suitable client or directly with the Open API) or can place it at the server side.

Both options are valid but each of them drives to different development styles. Having said this, it is important to note that in the WWW 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 this GE. That is, by locating the multimedia application logic at the server side, so that the specific media pipelines are created using the the client for your favorite language.

2.3.7 Main Interactions

Interactions from a generic perspective

A typical Stream Oriented GE application involves interactions among three main modules:
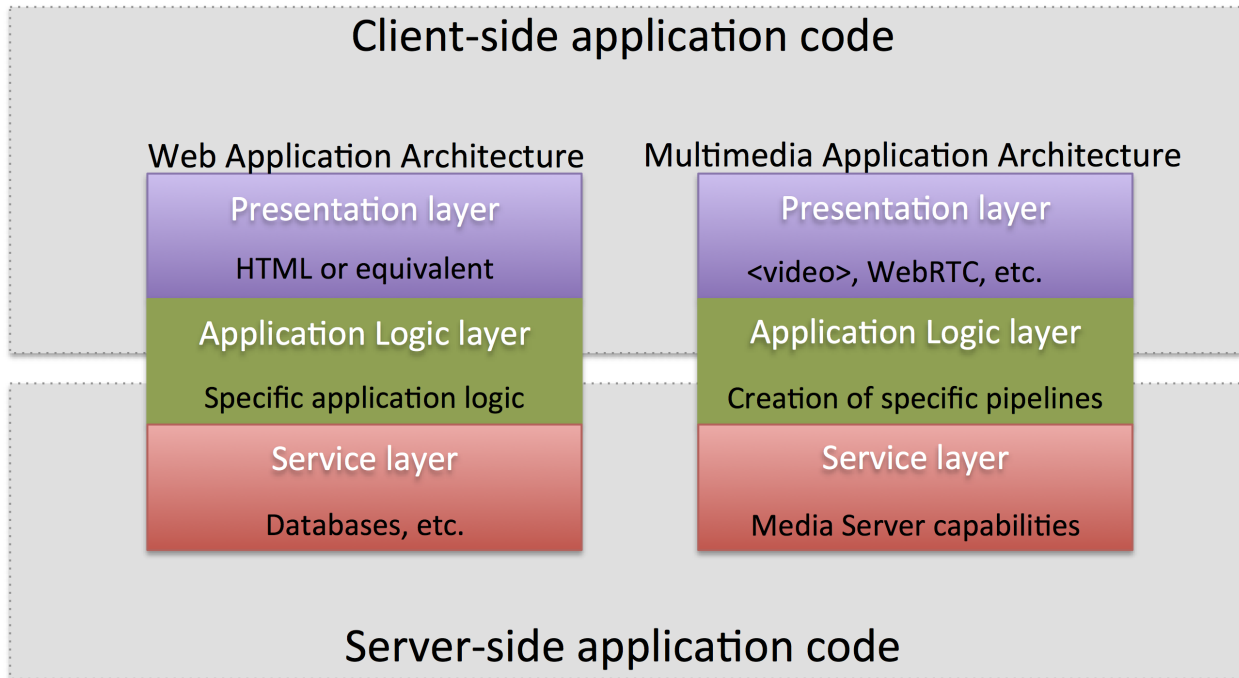
Fig. 2.5: *Applications created using the Stream Oriented GE (right) have an equivalent architecture to standard WWW applications (left). Both types of applications may choose to place the application logic at the client or at the server code.*

- **Client Application**: which involves the native multimedia capabilities of the client platform plus the specific client-side application logic. It can use Clients designed to client platforms (for example, JavaScript Client).

- **Application Server**: which involves an application server and the server-side application logic. It can use Clients designed to server platforms (for example, Java Client for Java EE and JavaScript Client for Node.js).

- **Media Server**: which receives commands for creating specific multimedia capabilities (i.e. specific pipelines adapted to the needs of specific applications)

The interactions maintained among these modules depend on the specificities of each application. However, in general, for most applications they can be reduced to the following conceptual scheme:

Media negotiation phase

As it can be observed, at a first stage, a client (a browser in a computer, a mobile application, etc.) issues a message requesting some kind of capability from the Stream Oriented GE. This message is based on a JSON RPC V2.0 representation and fulfills the Open API specification. It can be generated directly from the client application or, in case of web applications, indirectly consuming the abstract HTML5 SDK. For instance, that request could ask for the visualization of a given video clip.

When the Application Server 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 Server processes the request and, according to the specific instructions programmed by the developer, commands the Media Server to instantiate the suitable media elements and to chain them in an appropriate media pipeline. Once the pipeline has been created successfully the server responds accordingly and the Application Server forwards the successful response to the client, showing it how and where the media service can be reached.

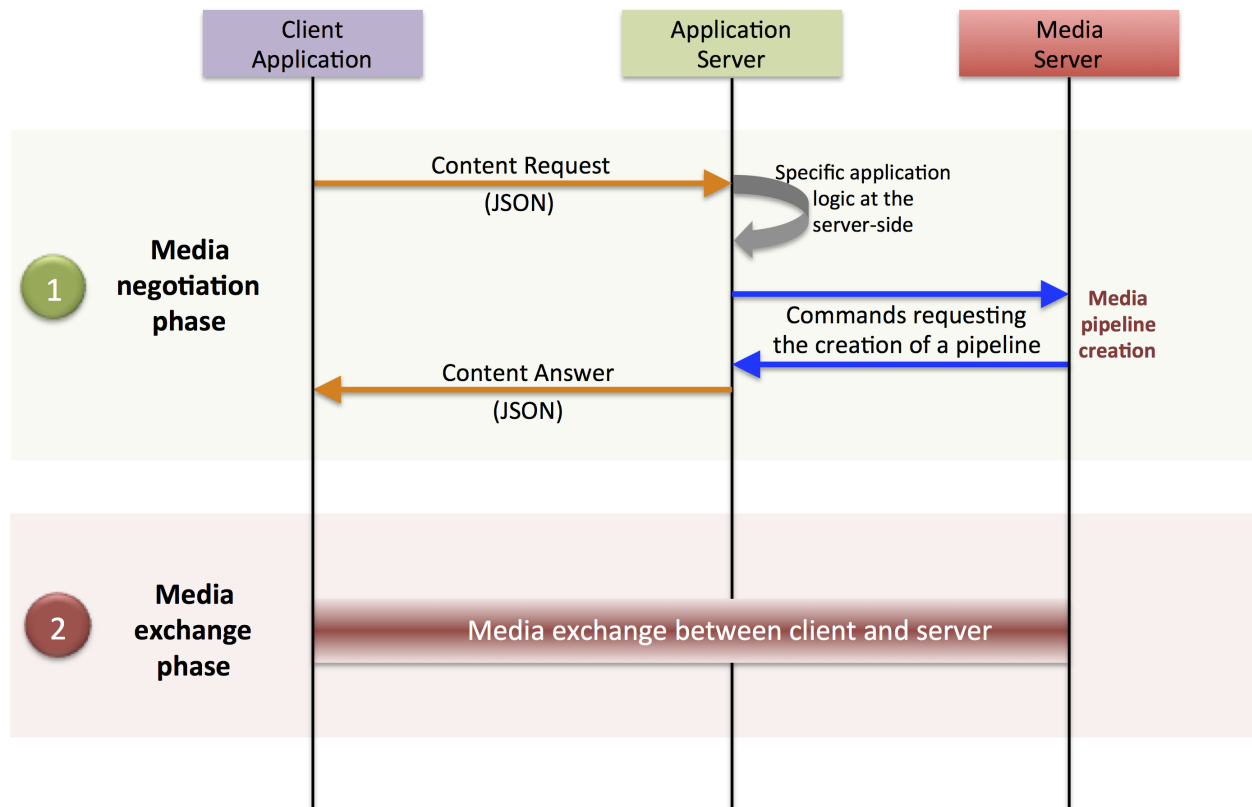Fig. 2.6: *Main interactions occur in two phases: negotiation and media exchange. Remark that the color of the different arrows and boxes is aligned with the architectural figures presented above, so that, for example, orange arrows show exchanges belonging to the Open API, blue arrows show exchanges belonging to the Thrift API, red boxes are associated to the Media Server and green boxes with the Application Server.*

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.

Media exchange phase

After that, a new phase starts devoted to producing the actual media exchange. The client addresses a request for the media to the 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 Media Server where the clip can be obtained and, as a result, an HTTP request with 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. The Stream Oriented GE 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 element during the negotiation phase. After that, from the client perspective, the processed clip will be received as any other video.

Specific interactions for commonly used services

Regardless of the actual type of session, all interactions follow the pattern described in section above. However, most common services respond to one of the following two main categories:

RTP/WebRTC

The Stream Oriented GE allows the establishment of real time multimedia session between a peer client and the Media Server directly through the use of RTP/RTCP or through WebRTC. In addition, the Media Server can be used to act as media proxy for making possible the communication among different peer clients, which are mediated by the Stream Oriented GE infrastructure. Hence, the GE can act as a conference bridge (Multi Conference Unit), as a machine-to-machine communication system, as a video call recording system, etc. As shown in the picture, the client exposes its media capabilities through an SDP (Session Description Protocol) payload encapsulated in a JSON object request. Hence, the Application Server is able to instantiate the appropriate media element (either RTP or WebRTC end points), and to require it to negotiate and offer a response SDP based on its own capabilities and on the offered SDP. When the answer SDP 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

As with the rest of examples shown above, 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. Just as an example, imagine that we 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.

HTTP recorder

HTTP recording sessions are equivalent to playing sessions although, in this case, the media goes from the client to the server using POST HTTP method. The negotiation phase hence starts with the client requesting to upload the content and the Application Server creating the appropriate pipeline for doing it. This pipeline will always start with an *HttpPostEndpoint* element. Further elements can be connected to that endpoint for filtering media, processing it or
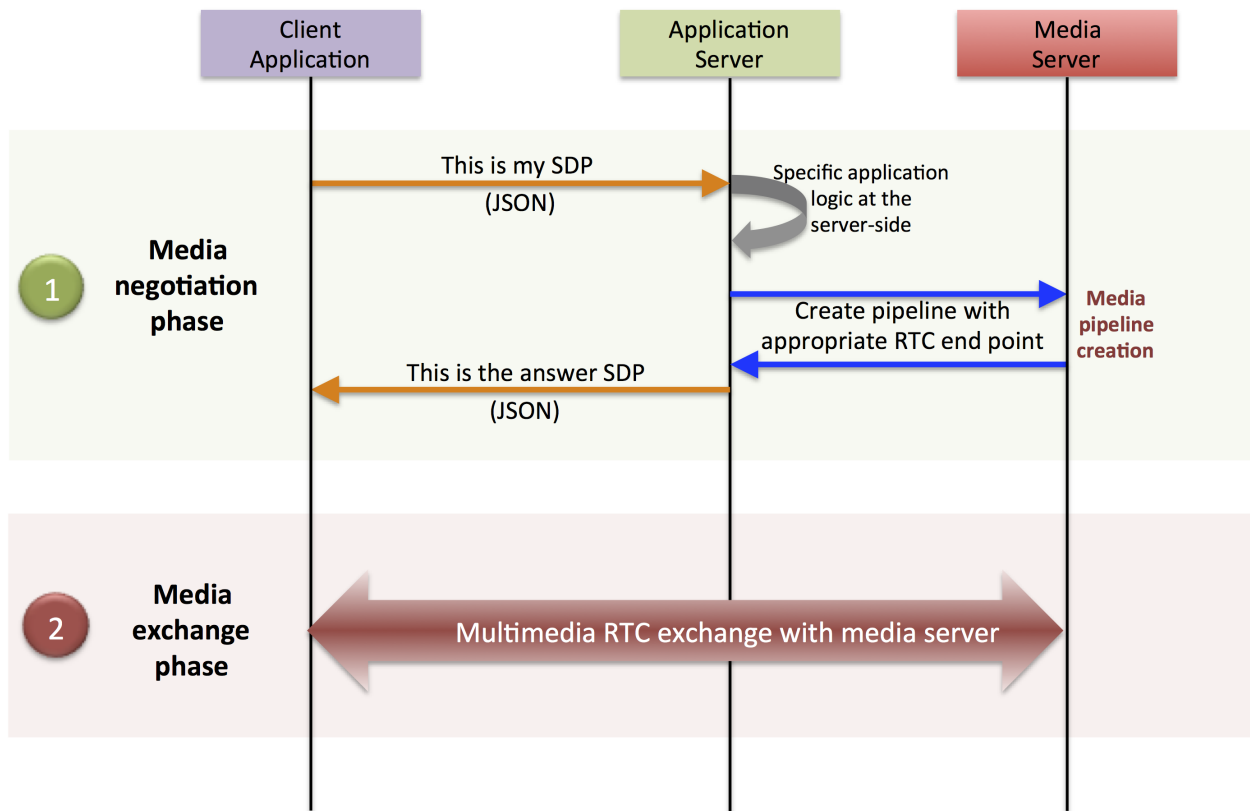
Fig. 2.7: *Interactions taking place in a Real Time Communications (RTC) session. During the negotiation phase, a SDP message is exchanged offering the capabilities of the client. As a result, the Media Server generates an SDP answer that can be used by the client for establishing the media exchange.*
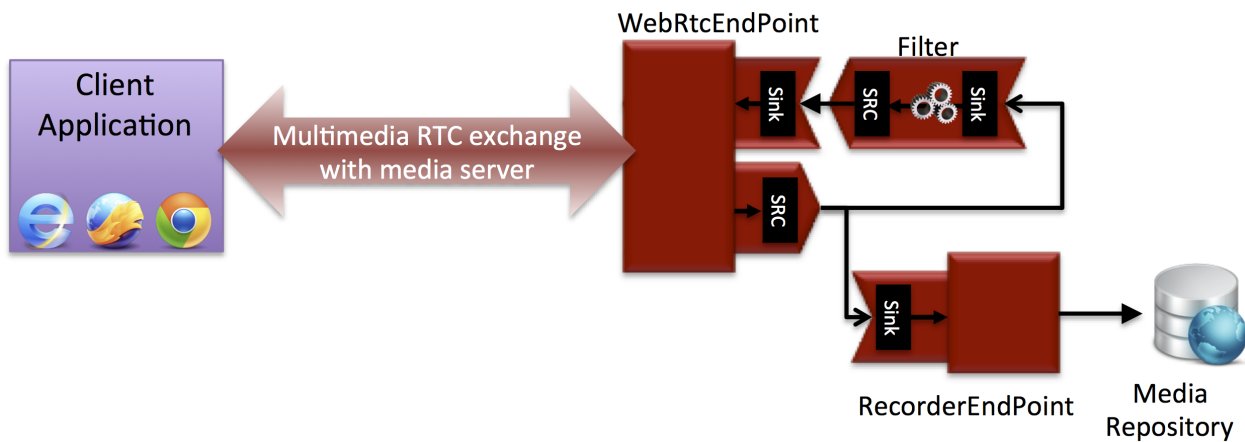
Fig. 2.8: *During the negotiation phase, the application developer can create a pipeline providing the desired specific functionality. For example, this pipeline uses a WebRtcEndpoint for communicating with the client, which is connected to a RecorderEndpoint storing the received media stream and to 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).*

storing it into a media repository. The specific interactions taking place in this type of session are shown in the figure below

Fig. 2.9: *Example of pipeline for an HTTP recorder*

Basic Design Principles

The Stream Oriented GE is designed based on the following main principles:

- *Separate Media and Signaling Planes*. Signaling and media are two separate planes and therefore the Stream Oriented GE is designed in a way that applications can handle separately those facets of multimedia processing.
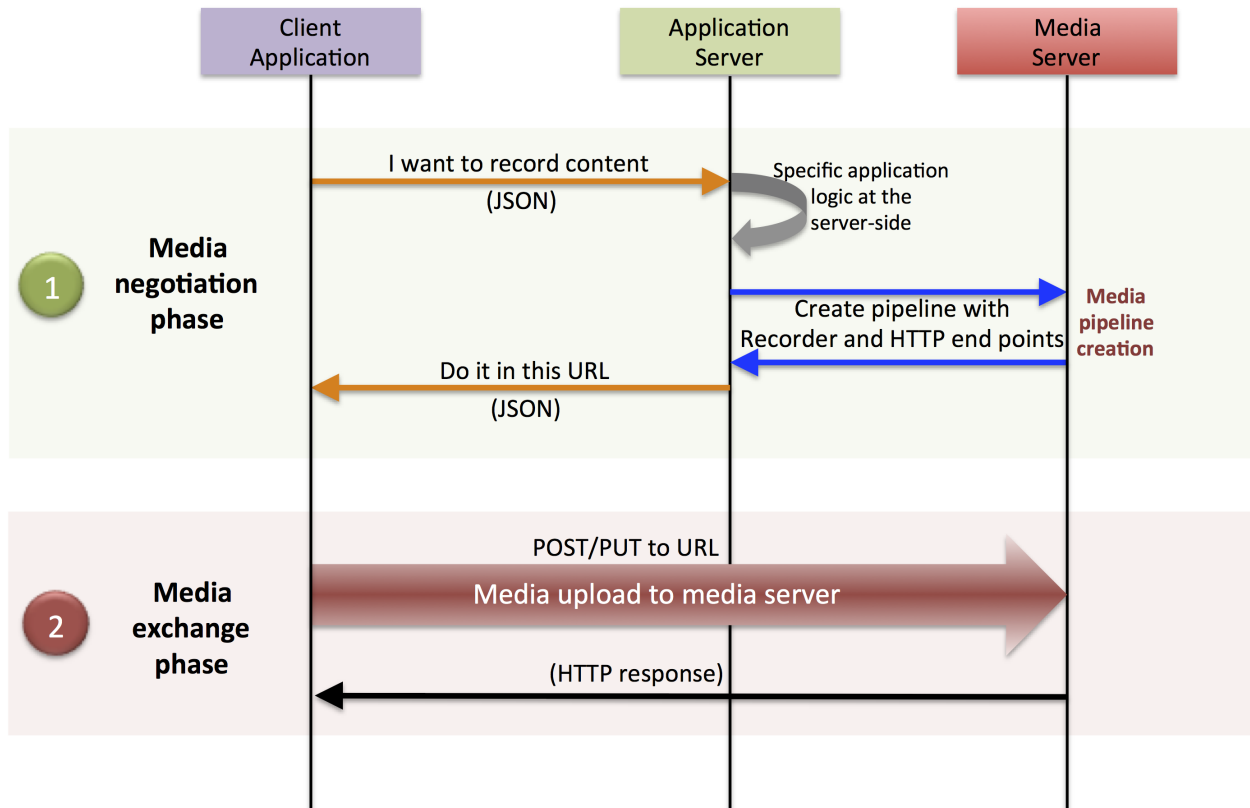
- *Distribution of Media and Application Services*. Media Server and applications can be collocated, escalated or distributed among different machines. A single application can invoke the services of more than one Media Servers. The opposite also applies, that is, a Media Server can attend the requests of more than one application.

- *Suitable for the Cloud*. The Stream Oriented GE is suitable to be integrated into cloud environments to act as a PaaS (Platform as a Service) component.

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

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

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

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

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

- *Auditable Processing.* The Stream Oriented GE is able to generate rich and detailed information for QoS monitoring, billing and auditing.

- *Seamless IMS integration.* The Stream Oriented GE is designed to support seamless integration into the IMS infrastructure of Telephony Carriers.

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

2.4 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*).

2.4.1 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.

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)

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

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)

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

2.4.2 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.

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 Media-Source 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)

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

```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"
}
```

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)

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

2.4.3 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.

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
        "operation": "connect",
        "operationParams": {
            "sink": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline/087b7777-aab5-4787-81
        },
        "sessionId": "bd4d6227-0463-4d52-b1c3-c71f0be68466"
    },
    "jsonrpc": "2.0"
}
```

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"
}
```

2.4.4 Release

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

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"
}
```

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"
}
```

2.4.5 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.

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)

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

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)

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

2.4.6 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.

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

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"
}
```

2.4.7 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.

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/76dcb8d7-5655-4
                "tags": [],
                "timestamp": "1461589478",
                "type": "EndOfStream"
            },
            "object": "6ba9067f-cdcf-4ea6-a6ee-d74519585acd_kurento.MediaPipeline/76dcb8d7-5655-445b-
            "type": "EndOfStream"
        }
    }
}
```

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

Response

There is no response to the *onEvent* message.