D4.1.1	
Version	1.0
Author	NAEVATEC
Dissemination	PU
Date	27/01/2015
Status	Final

D4.1: Distributed Media Pipeline Middleware v1

Project acronym:	NUBOMEDIA
Project title:	NUBOMEDIA: an elastic Platform as a Service (PaaS) cloud for interactive social multimedia
Project duration:	2014-02-01 to 2016-09-30
Project type:	STREP
Project reference:	610576
Project web page:	http://www.nubomedia.eu
Work package	WP4
WP leader	Javier López
Deliverable nature:	Prototype
Lead editor:	Javier López
Planned delivery date	01/2015
Actual delivery date	27/01/2015
Keywords	Kurento, media server, pipeline, media capabilities

The research leading to these results has been funded by the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement nº 610576

FP7 ICT-2013.1.6. Connected and Social Media

Contributors:

Javier Lopez (NAEVATEC)
Ivan Gracia (NAEVATEC)
José A. Santos Cadenas (NAEVATEC)
Miguel París Díaz (URJC)
David Fernández López (NAEVATEC)
Boni Garcia (URJC)

Internal Reviewer(s):

Luis Lopez (URJC)
Markus Ylikerälä (VTT)

Version History

Version	Date	Authors	Comments
0.1	25-09-2014	F. Javier Lopez	Initial Version
0.2	15-11-2014	F. Javier Lopez	Added SoTA
0.3	17-11-2014	F. Javier Lopez	Added introduction
0.4	22-12-2014	F. Javier Lopez	Added Architecture and Software Analysis sections
0.5	02-01-2015	F. Javier Lopez	Re-structured for final version of the deliverable.
0.6	15-01-2015	Ivan Gracia	Added content to Architecture section from several authors
1.0	16-01-2015	Luis Lopez	Generate final version for review

Table of contents

1 Executive summary 9

2 Introduction 9
2.1 State-of-the-art on RTC Media Server technologies 9
2.2 GStreamer 11
2.3 Kurento Media Server 13

3 Objectives 17

4 Implementation strategy 18
4.1.1 Required actions to extend and adapt GStreamer 18
4.1.2 Required actions to extend and adapt Kurento Media Server 19

5 Kurento Media Server new high-level architecture 19

6 Software architecture 21
KmsElement 21
6.1.1 21
6.1.2 Agnostic bin 22
6.1.3 Hubs 23
6.1.4 Kurento Media Server Modules 26
6.1.5 Distributed garbage collector 29
6.1.6 Kurento WebRtc and RTP Endpoints 29
6.1.7 Kurento PlayerEndpoint 32
6.1.8 Kurento RecorderEndpoint 33
6.1.9 Kurento HttpEndpoints 34

7 Implementation status (NUBOMEDIA Release 3) 36
7.1 Licensing models and restrictions 36
7.2 Obtaining the source code 36
7.3 Obtaining the documentation 37
7.4 Installation and configuration guide 37
7.4.1 Installing Kurento Media Server 37
7.4.2 Kurento Media Server Configuration 37
7.4.3 Installing Kurento Media Server behind a NAT 38

8 References 40

9 Annex I: Kurento.org public documentation 41

10 Annex II: JSON-RPC protocol 41
10.1 Create 41
10.2 Invoke 42
10.3 Release 43
10.4 Subscribe 43
10.5 Unsubscribe 44
10.6 onEvent 44
10.7 Error responses 45

11 Annex III: Kurento Module Definition 45

12 Annex IV: Kurento Module Creation Procedure 48
12.1 How to Develop Kurento Modules 50
12.2 OpenCV module 50
12.3 GStreamer module 50
12.4 For both kind of modules 51
12.5 Examples 52

List of Figures:

Figure 1: RTC applications, in general, and WebRTC applications, in particular, may use two different models. As shown at the top, the peer-to-peer model is based on direct communication among clients. This model provides minimum complexity and latency, but it also has important limitations. At the bottom, the infrastructure-mediated model, where a media server is mediating among the communicating clients. This model has higher latency and complexity, but it makes possible to enrich RTC services with additional capabilities such as transcoding (i.e. interoperability), efficient group communications (i.e. MCU or SFU models), recoding and media processing. 10

Figure 2: Media capabilities provided by state-of-the-art media server include: transcoding (top), group communications (middle) and archiving (bottom). 11

Figure 3: Diagram showing a pipeline as a chain of connected media elements, where source pads are connected to sink pads indicating how media streams flow through the processing graph. This pipeline exemplifies an audio player where an encoded file is first decoded and later feed to the sound driver. 12

Figure 4: At the top, vision and capabilities of state-of-the-art RTC media servers. At the bottom, vision and capabilities of the Kurento Media Server. As it can be observed, Kurento Media Server incorporates advanced features for media processing including computer vision, augmented reality or any other kind of capability compatible with its flexible modular architecture. 13

Figure 5: Different programming models for applications using Kurento Media Server. At the top, application directly consuming Kurento Media Server capabilities from the client-side logic. In the middle, application based on a Java EE application server, where the media logic is held at the application server. At the bottom, application based on a Node.js application server, where the media logic is held at the application server. 14

Figure 6: Some media elements made available through the Kurento Media Server API. Application developers just need to connect the media elements they desire for creating their multimedia-enabled applications. 15

Figure 7: Example of KMS media pipeline providing WebRTC recoding capabilities through connecting a `WebRtcEndpoint` and a `RecorderEndpoint` media elements. 16

Figure 8: Kurento Media Server high-level architecture. 20

Figure 9. KmsElement UML diagram 21

Figure 10: KmsElement internal structure 22

Figure 11: KmsAgnosticBin UML diagram 23

Figure 12: KmsAgnosticBin internal structure 23

Figure 13: KmsBaseHub UML diagram 24

Figure 14: Composite internal structure 25

Figure 15: Dispatcher internal structure 25

Figure 16: Alpha Blending internal structure 26

Figure 17: Kurento modules architecture is based on three different type of modules: core modules, which are distributed off the self with Kurento, built-in modules, which need to be downloaded and installed independently but are maintained by the Kurento team, and custom modules, which are created and maintained by third party organizations. 28

Figure 18: WebRTC and RTP endpoints UML diagram 29

Figure 19: BaseRtpEndpoint internal structure 30

Figure 20: RtpEndpoint internal structure 31

Figure 21: WebRtcEndpoint internal structure 31

Figure 22: KmsPlayerEndpoint UML diagram 33

Figure 23: KmsRecorderEndpoitn internal structure *33*

Figure 24: KmsRecorderEndpoint UML diagram *33*

Figure 25: KmsRecorderEndpoint internal structure *34*

Figure 26: KmsHttpEndpoint UML diagram *34*

Figure 27: KmsHttpGetEndpoint internal structure *35*

Figure 28: KmsHttpPostEndpoint internal structure *35*

Figure 29: Typical scenario of Kurento Media Server behind a NAT *39*

Figure 30: Kurento modules architecture. Kurento Media Server can be extended with built-in modules (crowddetector, pointerdetector, chroma, platedetector) and also with other custom modules *49*

Figure 31**: **Extended Kurento Toolbox. The basic Kurento toolbox (left side of the picture) is extended with more computer vision and augmented reality filters (right side of the picture) provided by the built-in modules. *49*

Acronyms and abbreviations:

API	Application Programming Interface
AR	Augmented Reality
IMS	IP Multimedia Subsystem
IoT	Internet of Things
KMS	Kurento Media Server
RTC	Real-Time Communications
RTP	Real-time Transport Protocol
SCTP	Stream Control Transmission Protocol
VCA	Video Content Analysis
WebRTC	Web Real Time Communications

1 Executive summary

This document contains a description of the Real-Time Communications (RTC) Media Server used by NUBOMEDIA. In the RTC jargon, the Media Server is understood as a number of software artifacts providing the low level media capabilities, which include media transport, media storage, media adaption and media processing. This NUBOMEDIA Media Server exposes these capabilities through an abstract API based on the concept of Media Pipelines: chains of media elements each of which provides a specific media functionality. Hence, the media pipeline defined a specific sequence of processing steps applied to media streams getting through the NUBOMEDIA media infrastructure.

The NUBOMEDIA Media Server has been created extending and modifying the Kurento Media Server (KMS): an Open Source Software Media Server based on the GStreamer software stack. Kurento Media Server acts as a container of modules providing the different NUBOMEDIA processing capabilities (i.e. Video Content Analysis, Augmented Reality, etc.)

2 Introduction

2.1 State-of-the-art on RTC Media Server technologies

In the context of RTC systems, a media server is a software stack (i.e. a subsystem) that provides media capabilities. Media capabilities are all the features related to the media itself (i.e. related with the bits of information representing audio and video). These may include:

- Media transport following specific RTC protocols
- Media recording and recovery
- Media transcoding
- Media distribution
- Media analysis
- Media augmentation and transformation
- Media mixing
- Etc.

In the RTC framework, sometimes media servers are frequently named depending on the specific capabilities they provide. For example, we speak about streaming media servers when dealing with servers capable of streaming multimedia content, or we say recording media server when we have the capability of recording the audio and video received, etc.

The concept of media server is quite old and has been used in the RTC literature for long time. For example, in the area of IMS standards, media servers were introduced as the Media Resource Function (MRF), which provides to IMS systems capabilities such as group communications, transcoding, recording, etc.

In the last few years, RTC media servers are living again a new gold era due to the emergence of WebRTC[1] technologies. Nowadays, WebRTC is the main trending topic in the multimedia RTC area and this is why WebRTC needs to be supported by any RTC media server wishing to play a role in the market. WebRTC services commonly require the presence of media servers, which are very useful when creating services beyond the standard WebRTC peer-to-peer call model. Some common examples of server-side media plane elements that could be used in WebRTC include:

- WebRTC media gateways – typical on services requiring protocol or format adaptations (as happens when integrating WebRTC with IMS)
- Multi Point Control Units (MCUs) and Selective Forwarding Units (SFUs) – used to support group communications
- Recording media servers – helpful when one needs to persist a WebRTC call
- Etc.

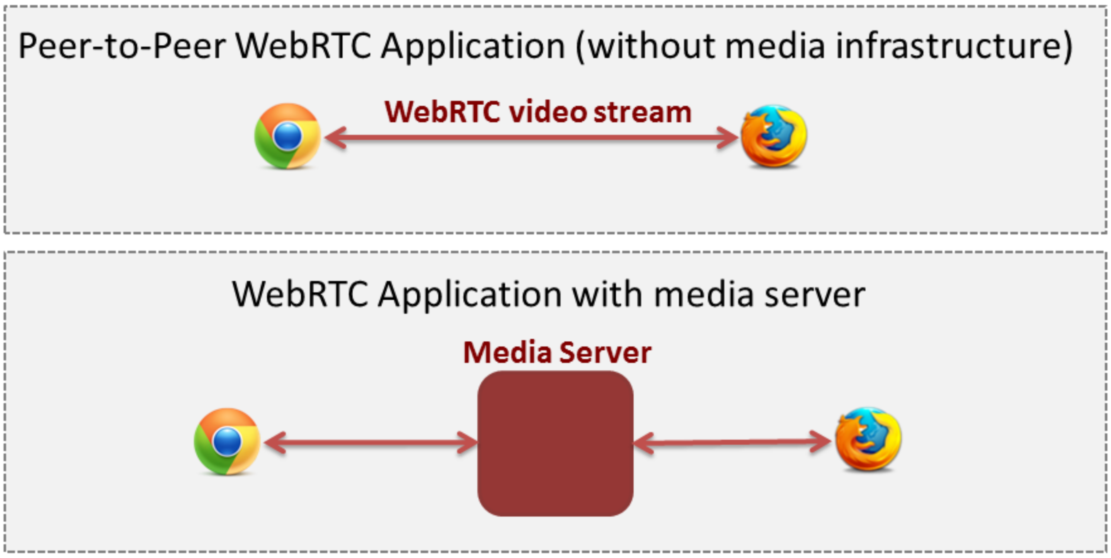
Peer-to-Peer WebRTC Application (without media infrastructure)
WebRTC video stream
WebRTC Application with media server
Media Server

Figure 1: RTC applications, in general, and WebRTC applications, in particular, may use two different models. As shown at the top, the peer-to-peer model is based on direct communication among clients. This model provides minimum complexity and latency, but it also has important limitations. At the bottom, the infrastructure-mediated model, where a media server is mediating among the communicating clients. This model has higher latency and complexity, but it makes possible to enrich RTC services with additional capabilities such as transcoding (i.e. interoperability), efficient group communications (i.e. MCU or SFU models), recoding and media processing.

Currently, there are a large number of WebRTC capable media servers out there. Just for illustration, the following is a non-exhaustive list containing some of the most common open source software solutions:

- Jitsi videobridge: An open source video bridge which is part of the Jitsi project. It implements SFU [2] capabilities useful for providing efficient and low latency group communications.
- Lynckia: An open source project that, as Jisti, provides an efficient SFU for group communications with some additional features such as WebM recording.
- MCU Media Server: An open source media server providing different capabilities which include transcoding, group communications through a mixing MCU [2], recording, etc.
- Janus: An open source media server providing different capabilities which include transcoding, group communications through an SFU model, recording, etc.
- telepresence: The open source media server of the Doubango initiative which provides group communications through a MCU mixing model and some additional features which include recording, 3D sound, etc.

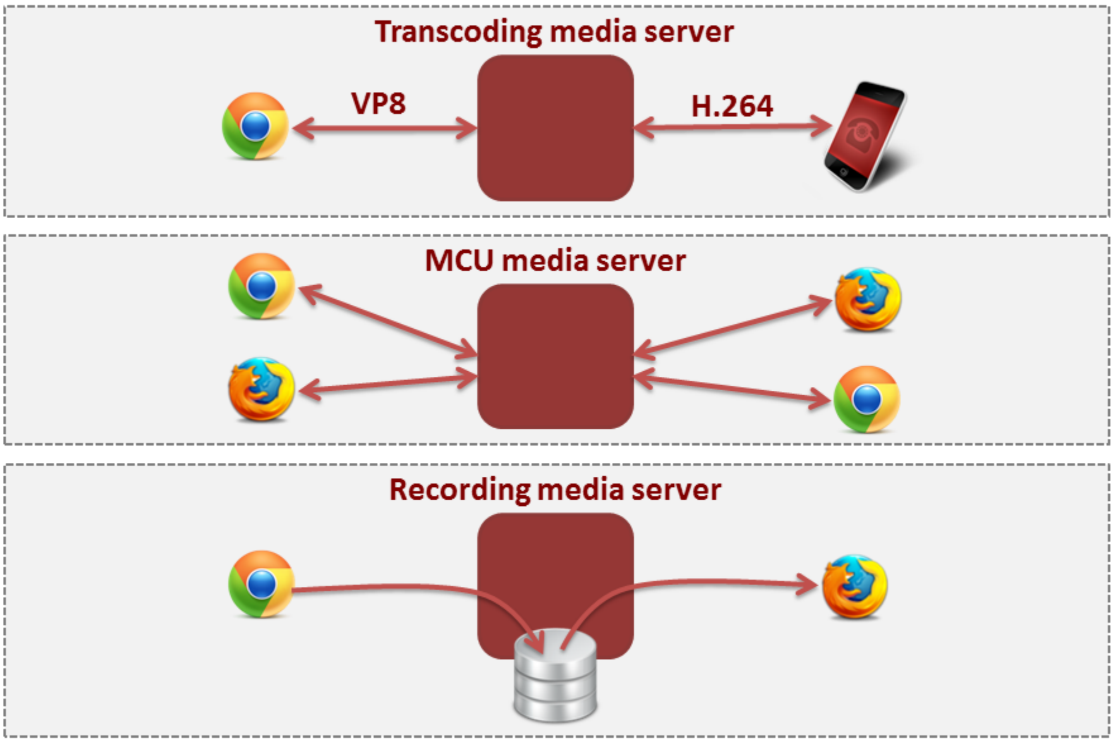

Figure 2: Media capabilities provided by state-of-the-art media server include: transcoding (top), group communications (middle) and archiving (bottom).

However, as NUBOMEDIA's own vision shows, there are many interesting things we can do with the media beyond the basic three capabilities enumerated above (i.e. transcoding, group communications, recording). These capabilities could, for example, enrich media with augmented reality, analyze media with computer vision and deep speech analysis, and finally, modify media with blend, replicate, special effects etc. These kinds of capabilities might provide differentiation and added value to applications in many specific verticals including e-Health, e-Learning, security, entertainment, games, advertising or CRMs just to cite a few.

Due to this, NUBOMEDIA requirements cannot be satisfied with any of the above mentioned solutions. Hence, for the execution of the project we need a more flexible technology where advanced processing capabilities could be plugged seamlessly. For this, we have re-architected the only open source media server enabling such types of capabilities: Kurento Media Server (KMS). Sections below describe in detail what KMS is and how we are using it.

2.2 GStreamer

GStreamer is an open source software project providing a pipeline-based multimedia framework written in the C programming language with the type system based on GObject. GStreamer is at the nucleus of Kurento Media Server (KMS), so that KMS can be seen as a wrapper for GStreamer for creating a media server abstraction on top of it and for exposing a coherent and simple to use API for developers. For this reason, all the advantages, features and limitations of GStreamer are inherited by KMS.

GStreamer is subject to the terms of the GNU Lesser General Public License (LGPL) and is currently supported in all major operating systems including

- BSD

- Linux
- OpenSolaris
- Android
- Maemo
- OS X
- iOS
- Windows
- OS/400

The GNOME desktop environment is a heavy user of GStreamer and includes it since GNOME version 2.2

GStreamer processes media by connecting a number of processing elements into a pipeline. Each element is provided by a plug-in. Elements can be grouped into bins (i.e. chains of elements), which can be further aggregated, thus forming a hierarchical graph, which follows the model of filter graphs shown in Fig. 3. In the context of multimedia processing, a filter graph is a directed graph where edges represent one-way data flows and nodes represent a data processing step. The term pin or pad is used to describe the connection point between nodes and edges, so that a source pad in one element can be connected to a sink pad on another.

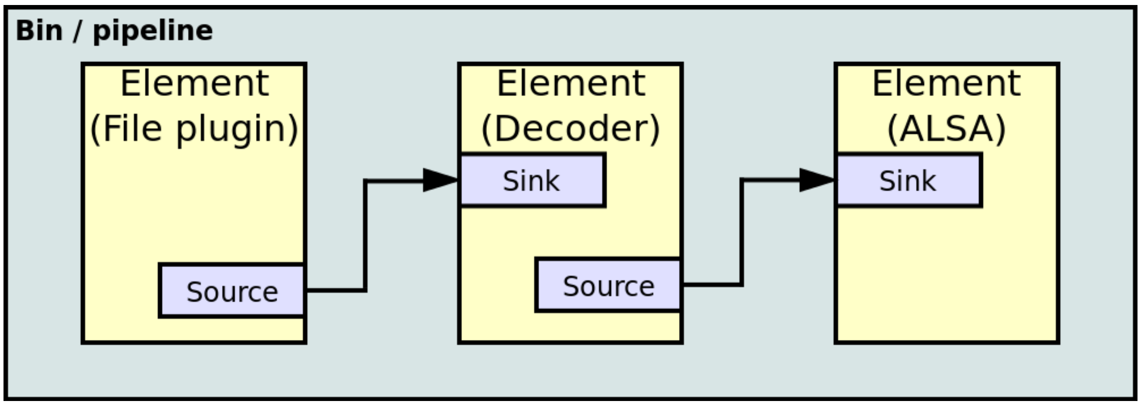

Figure 3: Diagram showing a pipeline as a chain of connected media elements, where source pads are connected to sink pads indicating how media streams flow through the processing graph. This pipeline exemplifies an audio player where an encoded file is first decoded and later feed to the sound driver.

When the pipeline is in the playing state, data buffers flow from the source pad to the sink pad. Pads negotiate the kind of data that will be sent using capabilities.

GStreamer uses a plug-in architecture based on shared libraries. Plug-ins can be loaded dynamically providing support for a wide spectrum of codecs, container formats, drivers and effects.

Plug-ins can be installed semi-automatically when they are first needed. For that purpose, distributions can register a backend that resolves feature-descriptions to package-names. Plug-ins are grouped in three sets: the Good, the Bad and the Ugly

- Good: a set of well-supported and high-quality plug-ins under LGPL license.
- Bad: plug-ins that might closely approach good-quality, but they lack something (e.g. a good code review, some documentation, tests, a maintainer, etc.)
- Ugly: a set of good-quality plug-ins that might pose distributions problems (i.e. licensing issues, IPR issues, etc.)

The Good, the Bad and the Ugly plug-ins provide support for a wide variety of media formats, codec, protocols and containers including

- MPEG-1
- MPEG-2
- MPEG-4
- H.261
- H.263
- H.264
- RealVideo
- MP3
- WMV
- FLV
- VP8
- VP9
- Daala
- Etc.

At the time of this writing, current stable GStreamer release is 1.4.5, which among other features, provides the following capabilities with relevance for NUBOMEDIA:

- Hardware accelerated video encoding/decoding using GPUs
- Dynamic pipelines (i.e the capability of modifying the processing graph dynamically at any time during the lifecycle of the multimedia application)

2.3 Kurento Media Server

Kurento Media Server (KMS) is an RTC media server and a set of client APIs simplifying the development of advanced multimedia applications for WWW and smartphone platforms. KMS features include group communications, transcoding, recording, mixing, broadcasting and routing of audiovisual flows.

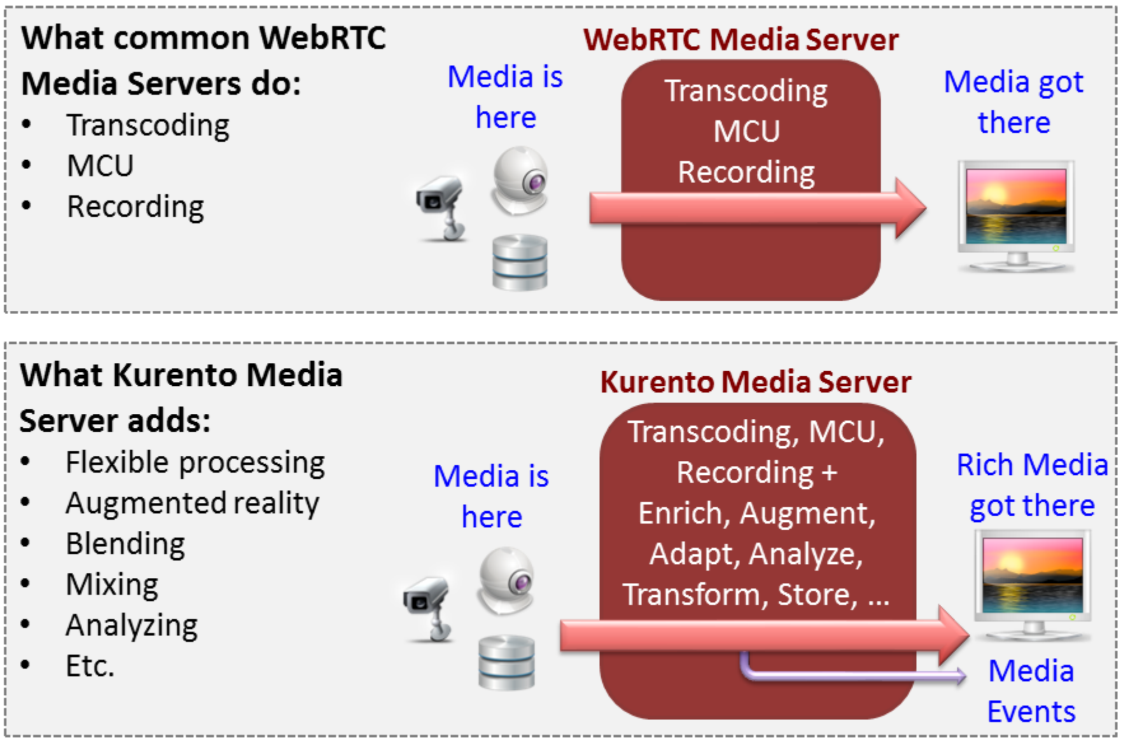

Figure 4: At the top, vision and capabilities of state-of-the-art RTC media servers. At the bottom, vision and capabilities of the Kurento Media Server. As it can be observed, Kurento Media Server

incorporates advanced features for media processing including computer vision, augmented reality or any other kind of capability compatible with its flexible modular architecture.

As a differential feature, KMS also provides advanced media processing capabilities involving computer vision, video indexing, augmented reality and speech analysis. Kurento 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 Kurento built-in features.

Kurento Media Server has been created as a join effort between Universidad Rey Juan Carlos (URJC) and Naevatec through the execution of several industrial and research projects including NUBOMEDIA, FI-WARE, FI-CORE and Aficus.

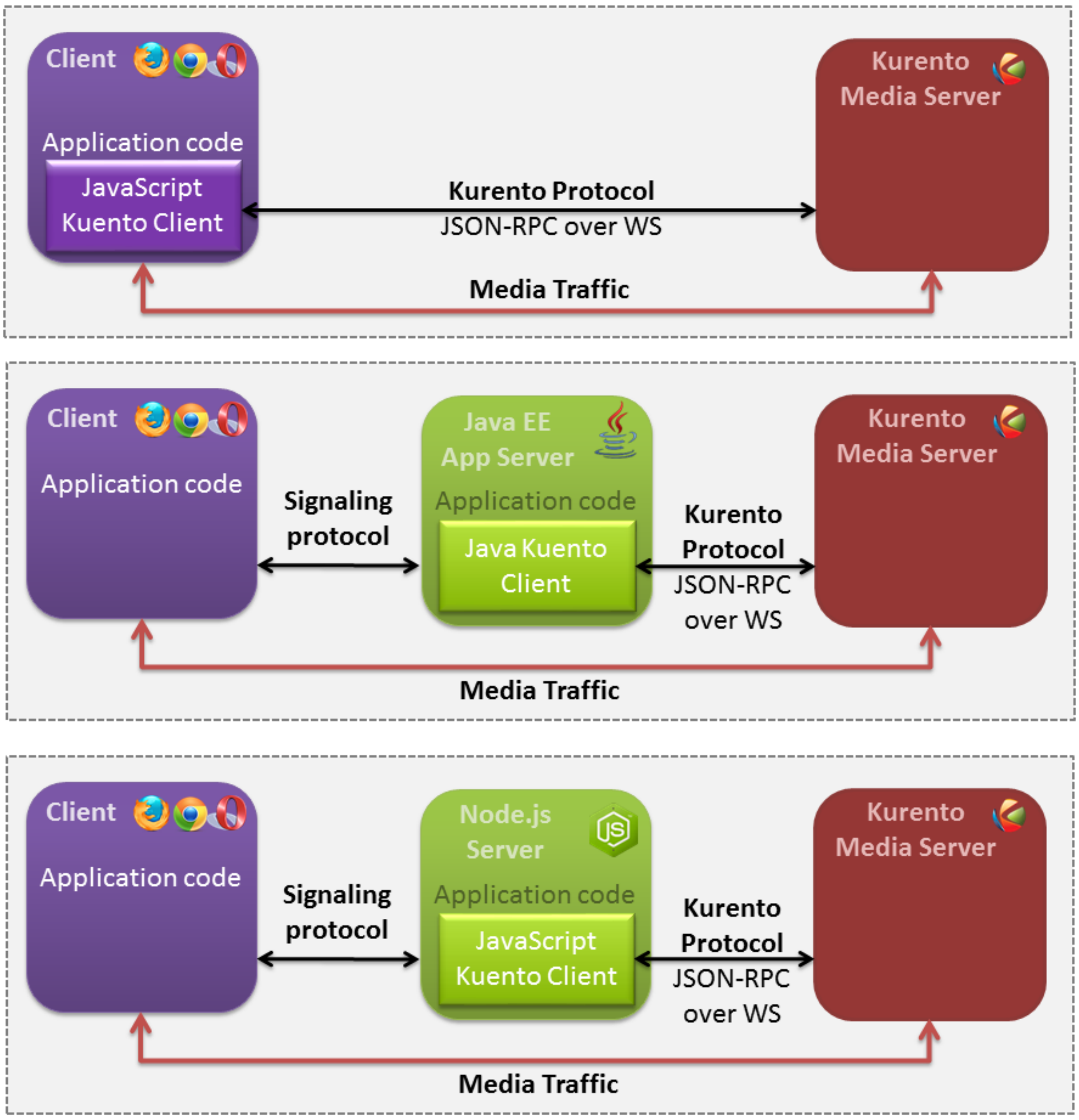

Figure 5: Different programming models for applications using Kurento Media Server. At the top, application directly consuming Kurento Media Server capabilities from the client-side logic. In the middle, application based on a Java EE application server, where the media logic is held at the application server. At the bottom, application based on a Node.js application server, where the media logic is held at the application server.

Kurento Media Server capabilities are exposed by the Kurento API to application developers. This API is implemented by means of libraries called Kurento Clients.

Kurento offers two clients out of the box for Java and JavaScript. If you have another favorite language, you can still use Kurento using directly the Kurento Protocol. This protocol allows to control Kurento Media Server and it is based on Internet standards such as WebSocket and JSON-RPC. The picture below shows how to use Kurento Clients in three scenarios:

- Using the Kurento JavaScript Client directly in a compliant WebRTC browser
- Using the Kurento Java Client in a Java EE Application Server
- Using the Kurento JavaScript Client in a Node.js server

Kurento Client's API is 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.

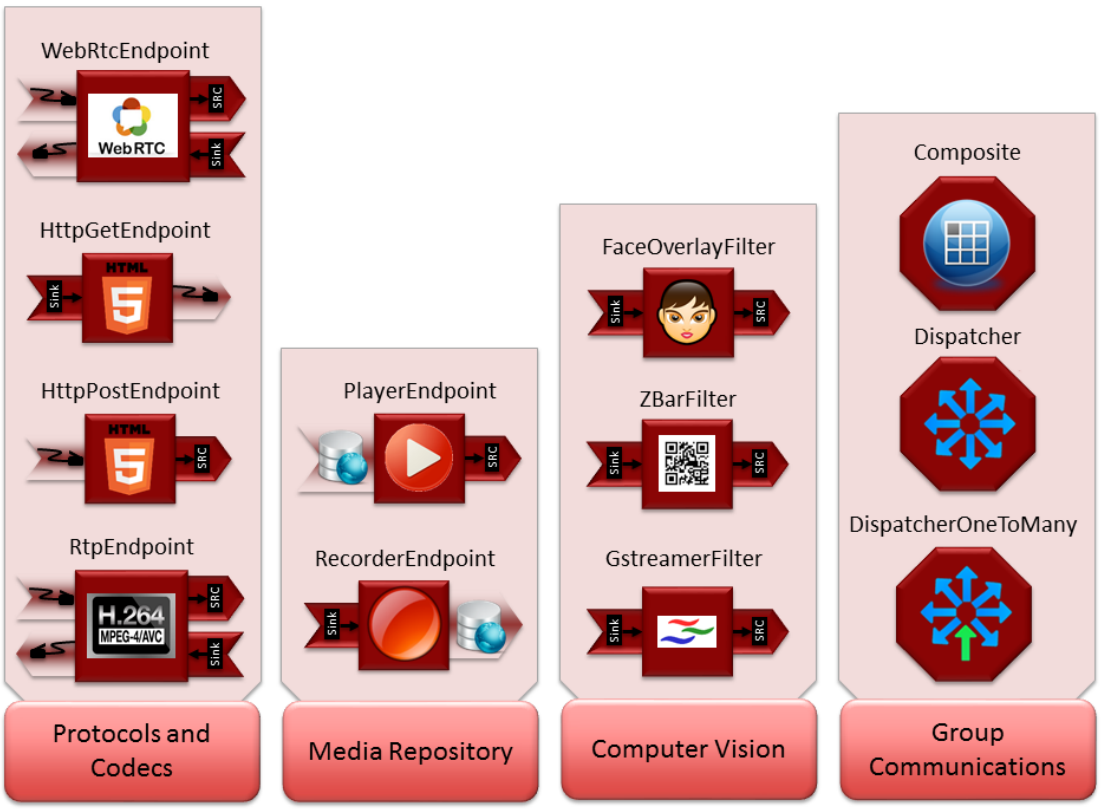

Figure 6: Some media elements made available through the Kurento Media Server API. Application developers just need to connect the media elements they desire for creating their multimedia-enabled applications.

From the application developer perspective, Media Elements are like Lego pieces: you just need to take the elements needed for an application and connect them following the desired topology. In Kurento jargon, a graph of connected media elements is called a Media Pipeline. Hence, when creating a pipeline, developers need to determine the capabilities they want to use (the media elements) and the topology determining which media 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)
```

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 fed into the later:

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

To simplify the handling of WebRTC streams in the client-side, Kurento provides a utility called `WebRtcPeer`. Nevertheless, the standard WebRTC API (`getUserMedia`, `RTCPeerConnection`, and so on) can also be used to connect to `WebRtcEndpoints`.

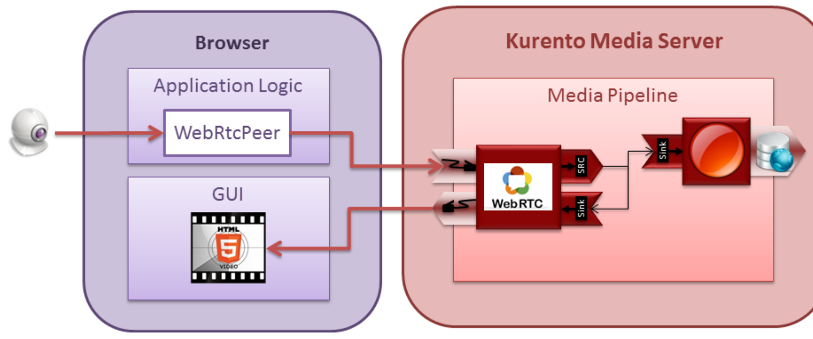

Figure 7: Example of KMS media pipeline providing WebRTC recoding capabilities through connecting a `WebRtcEndpoint` and a `RecorderEndpoint` media elements.

Following this description, it is straightforward to understand the relation between the GStreamer pipeline-model and the KMS pipeline model, given that both are based on the filter graph model: atomic (i.e. unsplittable) processing units called media elements that are chained through a directed graph to create a full and customized processing pipeline. However, from the perspective of developers, there are significant differences that are important to understand. KMS media elements have been designed for being simple to use by developers, who can connect and disconnect them safely at any time and without worrying for low level details such as codecs, formats, frames or capabilities. These differences can be summarized in the following way:

- KMS media elements are composed of many GStreamer media elements, where some of these low-level media elements play relevant roles for simplifying the life of application developers.
- KMS media elements can be connected/disconnected dynamically, at any time and in a thread safe manner. GStreamer media elements cannot be dynamically connected/disconnected at any time and developers need to take into consideration very low level details of the codecs for being able to implement dynamicity (e.g. without additional logic, GStreamer re-connections can only be

executed in GoP boundaries just when buffers associated to key frames traverse pads.

- KMS media elements abstract for developers all aspects relative to codecs, so that transcoding happens automatically and transparently. GStreamer media elements require developers to insert into the pipeline the appropriate transcoding logic when different formats need to coexist in the pipeline.
- KMS media elements abstract the need of using GStreamer control capabilities including queues and valves, so that developers don't need to be aware of their existence. GStreamer media elements require from developers the need of inserting such types of capabilities when they are necessary for pipelines to work properly.
- KMS media elements provide a built-in one-to-many capability, so that a media element source pad can be connected to as many media element sink pads as you want. In GStreamer, one-to-many graphs require the use of specific media elements and bins that developers need to know and manage.

As a result, KMS can be seen as a wrapper to GStreamer, which extends and abstracts its capabilities and exposes them through a simple to use API suitable for being used by non-expert developers.

3 Objectives

The main objective of this deliverable, is to provide the NUBOMEDIA platform with a media function capability, that can be used for creating a cloud media plane with the appropriate media capabilities, and suitable for working with elastic scalability. The execution of this objective requires the fulfillment of a number of sub-objectives, which are the following:

- To create a software media server suitable for holding all NUBOMEDIA features, which include: RTC communication protocols (including WebRTC ones), group communications, transcoding, recoding, media playing, computer vision and augmented reality. These capabilities need to be presented as pluggable modules for the media server, so that additional modules can be plugged without requiring re-compiling and re-deploying the media server.
- To create the appropriate logic making possible to connect module instances (i.e. media elements) to form media processing pipelines following the filter graph model. This logic must enable the creation of arbitrary complex processing graphs (i.e. media pipelines) without restrictions in relation to the number of sinks that can be connected to a given source (other than the own scalability restrictions of the host machine), without restrictions in the dynamicity of the links (so that edges can be added or removed at any time during the lifecycle of the pipeline), without constraints related to media formats (so that the connecting logic abstracts all aspects relative to codecs and formats, making the appropriate transcoding when required) and without constraints relative to the nature and origin of the media, so that developers don't need to be aware on aspects such as whether media is coming from a live source or a file, or whether it's coming from a lossy protocol or a lossless source.
- To create the appropriate interfaces making possible to abstract the low level details of the media server architecture to module creators. This means that developers need to be able to create additional modules in a seamless and direct manner without needing to understand all the details related to codecs, protocols, buffers or threads used by the media server.

- To create the appropriate logic making possible the media streams to maintain the appropriate quality levels. This include support for high quality codecs, support for synchronization mechanisms of audio and video tracks belonging to the same session, support for RTC protocols providing adaptive bandwidth, support for RTC protocols sensible to packet loss, support for low latency services, etc.
- To create the appropriate capabilities making possible the existence of distributed media pipelines where the different involved media elements may reside in different processes belonging to different computers in an IP network.
- To create the appropriate networking APIs making possible to control the media server behavior and capabilities from client applications.

4 Implementation strategy

The achievement of the above mentioned objectives cannot be accomplished by direct re-use of any of the pre-existing open source software solutions. For this, we need to go beyond state-of-the-art and create a new media server technology providing the appropriate features and capabilities.

The implementation strategy for this deliverable has been based on two main principles.

- To use GStreamer as a low-level media framework on top of which the required media capabilities will be implemented.
- To use Kurento Media Server as a top-level media framework for creating the appropriate abstractions, APIs and wrappers suitable for exposing the GStreamer-based low level capabilities to developers appropriately.

Neither GStreamer nor Kurento Media Server provide, off the shelf, the features required for NUBOMEDIA. For this reason, this strategy requires the execution of a number of developments, adaptations and extensions, which are summarized in the next subsections.

4.1.1 Required actions to extend and adapt GStreamer

GStreamer is a mature project, which has been around for more than 20 years now. For this reason, it already provides many of the low level features that we require in relation to the media infrastructure for holding media elements and connecting them as media pipelines. However, there are two major issues that need to be solved for being able to re-use GStreamer for our objectives:

- Instabilities due to lack of maintenance: GStreamer is not a mass-scale project and, for this reason, it is currently used only by some hundreds of developers, among which only some dozens work in maintaining its stability. This is particularly important when working with rare capabilities such as dynamic pipelines, which are not among the mainstream use models of the community. This makes necessary to include in the working plan of this deliverable the need to reserve efforts for working in the maintenance of such features, in collaboration with core GStreamer developers.
- Lack of latest trends capabilities: GStreamer evolution is quite slow, which makes it not to have ready many required capabilities associated to latest trends in RTC. In particular, before NUBOMEDIA, GStreamer did not provided any kind of WebRTC capabilities, so our implementation strategy needed to take into account this aspect when establishing the appropriate planning.

4.1.2 Required actions to extend and adapt Kurento Media Server

Kurento Media Server is a quite young project, which was created for providing arbitrary processing capabilities to real-time media streams. Using it for NUBOMEDIA requires taking into account the following actions and extensions:

- Instabilities and lack of maturity: KMS is barely two years old and its user base is still scarce. This makes KMS require continuous maintenance efforts for solving instabilities and problems originated by its lack of maturity. A relevant percentage of efforts that need to be invested for transforming KMS into the NUBOMEDIA media server have to be devoted to this.
- Lack of WebRTC capabilities: When NUBOMEDIA started, KMS WebRTC capabilities were too basic, without support for any kind of mechanism for dealing with non-optimal networks. Features such as appropriate QoS feedback (RFC 4585), congestion control and packet loss management were not in the original KMS status.
- Lack of distributed media pipelines capabilities: KMS had not build-in capabilities for creating distributed media pipelines. Basic mechanism for media distribution basing on the DGP protocol were available at GStreamer level, but this mechanism cannot comply with NUBOMEDIA requirements given that it does not provide a distributed media pipeline abstraction (i.e. GStreamer events are not propagated with the media), making it useless for NUBOMEDIA objectives.
- Lack of a modular architecture: In its original status, KMS had no modular architecture requiring a re-compilation and re-deployment of the whole media server every time a new module was added.
- Lack of efficiency: In its original status, KMS lacked the appropriate efficiency for working in a scalable way with media streams. Every media element required the consumption of around 15 threads, making it useless for any kind professional level application one could imagine.
- Lack of cloud integration capabilities: For being useful for NUBOMEDIA, KMS needs the appropriate mechanism for working into a cloud environment. In particular, it needs to implement the interfaces enabling its management and control from the NUBOMEDIA Elastic Media Manager, as it has been described in deliverable D3.5.1.
- Lack of monitoring capability: KMS did not provide any kind of monitoring capability, making it useless for implementing the auto-scaling mechanism required by NUBOMEDIA (as described in D3.5.1) and the CDR scheme NUBOMEDIA uses for billing and optimization (as described in D3.7.1).
- Lack of multisensory data capability: KMS did not provide any kind of mechanism for the transport and management of media data other than audio and video, nor for the synchronization of such data with the audiovisual flows.

In this context, this deliverable consists on implementing the strategy mentioned above for adapting and extending KMS. As a result, we have generated a novel KMS version, whose description is presented in the following sections.

5 Kurento Media Server new high-level architecture

Kurento Media Server current architecture is based on a layered model, as depicted in Figure 8. The different layers play specific roles:

- Transports: provide transports for the JSON-RPC Control Protocol. These transports constitute the northbound interface of the media server, and allow remote clients to connect and access KMS capabilities from other locations of the network. Currently, two types of transports are supported:

- WebSocket Transport: This transport is based on the WebSocket protocol, as specified in RFC 6455. Hence, KMS as a WebSocket server capable of accepting connections and establishing communications with WebSocket clients compatible with that standard.
- AMQP Transport: This transport is based on RabbitMQ AMQP implementation, and makes KMS behave as an AMQP client. As such, it can receive messages from other AMQP clients.

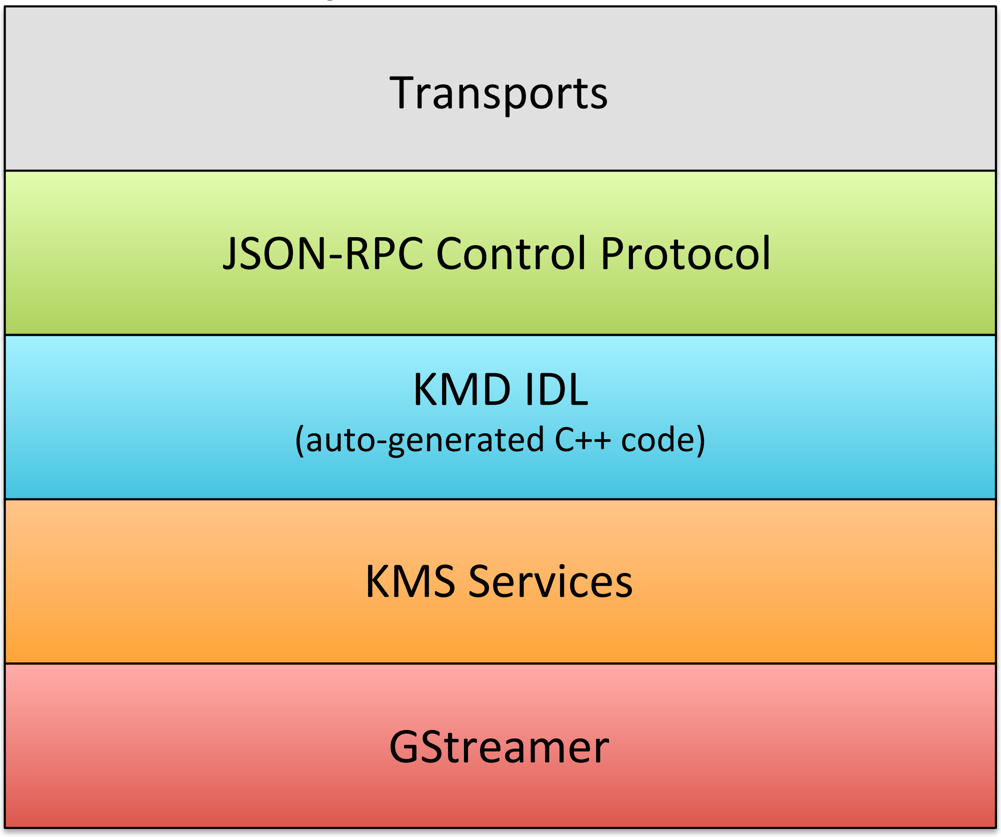

Figure 8: Kurento Media Server high-level architecture.

- JSON-RPC Control Protocol: clients, using any of the available transports, communicate with KMS and control KMS capabilities using a JSON-RPC v2.0 based protocol. This protocol is somehow similar to other media server control protocols such as MSML, but adapted to the specific requirements of NUBOMEDIA. The specification of this protocol can be found in Annex II: JSON-RPC protocol.
- KMD IDL: This is the nucleus of KMS modular architecture. It consists on an Interface Definition Language (IDL) based in JSON and specifically designed for the definition of KMS Modules. This language, called KMD (Kurento Module Definition) makes it possible for developers to define modules (media elements) through an interface (i.e. the specification of the commands that the media element accept and the events it generates). The specification of this language can be found in Annex III: Kurento Module Definition. Kurento module developers create new modules by defining the module interface in KMD. After that, this interface is compiled using a tool called kurento-module-creator, which generates the appropriate source code for integrating the module into KMS and for using the module from the different Kurento client libraries.

The process for creating a new module in specified in Annex IV: Kurento Module Creation Procedure.

- KMS Services: These comprise a number of services (i.e. primitives and boiler-plate code) that is used by the KMD compiled modules. These services provide the basic infrastructure of the media server in relation to thread and memory management, garbage collection, media element lifecycle management, media event management, etc.
- GStreamer: At the very bottom of KMS architecture we can find GStreamer capabilities, which are managed by the KMS Services layer in order to create the appropriate abstractions required by the modules (KMD IDL) layer. Hence, GStreamer capabilities cannot be accessed directly from application developers. This makes it possible to hide completely GStreamer complexities and to guarantee that developers can safely use GStreamer without requiring a deep understanding on low level details related to codecs, formats or negotiations.

6 Software architecture

6.1.1 KmsElement

This is the base class of all media elements. KmsElement inherits from GstBin (container), which in turn is a GstElement. This inheritance tree, shown in **¡Error! No se encuentra el origen de la referencia.** provides two key features: elements can have pads, and elements can contain other GStreamer elements (i.e. they are bins). Pads are an abstraction used in GStreamer to connect different elements, which means that a KmsElement can be connected to other KmsElements, thus creating a path over which the media is passed. On the other hand, as bins or containers, KmsElements can contain other GStreamer elements, which are really the ones in charge of processing media. This is the case of the Agnostic bin, a GStreamer element that exists in each KmsElement, in charge of managing the media transformations needed.

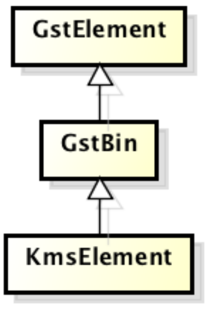

Figure 9. KmsElement UML diagram

The internal structure of a KmsElement, with the internal GStreamer elements that it contains, is shown in **¡Error! No se encuentra el origen de la referencia.**. There are different types of elements, color-coded for clarity.

- Violet boxes represent sink pads, GStreamer pads that receive media.
- Red boxes represent source pads, GStreamer pads that send media.
- Green boxes represent GStreamer elements.

- Grey "cloud" represents the part of the KMS element where the media processing takes place.

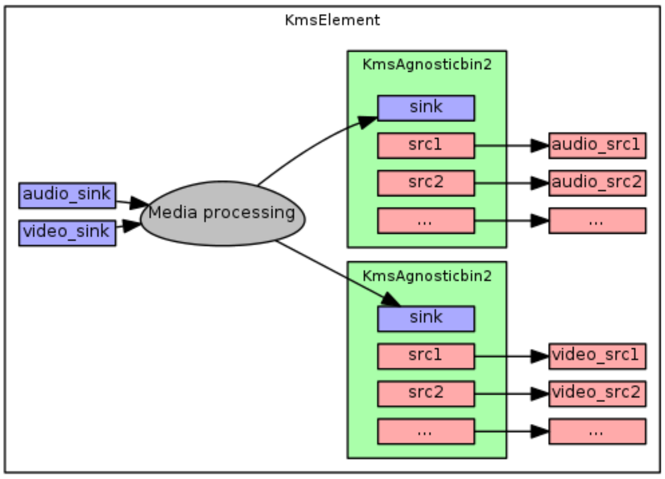

Figure 10: KmsElement internal structure

So from the previous figure, we can deduce that, after receiving audio and/or video through its sink pads, a media element will perform some media processing operations. When that processing is over, video and audio are sent separately to a special GStreamer element, the AgnosticBin, arriving at the bin's sink pads. This tailor-made element is in charge of adapting between different media formats, and performing any transcodifications needed by the prospective receivers of the processed media. The source pads of the AgnosticBin, are then connected with the source pads of the enclosing KmsElement as new consumers connect to receive the media.

This process is uniform to all KmsElements, and could be resumed as: after arriving in the KmsElement's sinks, the media is transformed, adapted in format and encoding to the needs of each consumer, and made available to them in the KmsElement's sources.

Another important feature that can be observed in the diagram, is that KmsElement only have one sink pad for audio and one sink pad for video, implying the only video and/or audio from one source can be received. There is a different kind of elements, called *Hubs*, which can receive media from more than one source, and they are explained in section 6.1.3. On the other hand, there is no limit as to how many source pads to serve media can a KmsElement have, thus being able to serve as many clients as needed, limited only by the computing resources available at any given time to the KmsElement.

It is the duty of each extending class to implement their own media processing transformations (the grey "cloud"), if needed. They must also signal when they are ready to receive audio and video, so the sink pads can be created, and to indicate the KmsElement parent which type of GStreamer element has to be created to receive the media. As an example, if the extending class expects to receive video in raw format (i.e. not encoded), it has to indicate so.

6.1.2 Agnostic bin

This element, as explained in the previous section, is responsible for the seamless interconnection of all media elements. It is thanks to this GStreamer element, that the programmer can safely ignore which formats are supported by each element, making it possible, for instance, to connect one WebRTC - VP8 encoded - endpoint, with and

RTC - H264 encoded - endpoint, without requiring any special actions from the programmer.

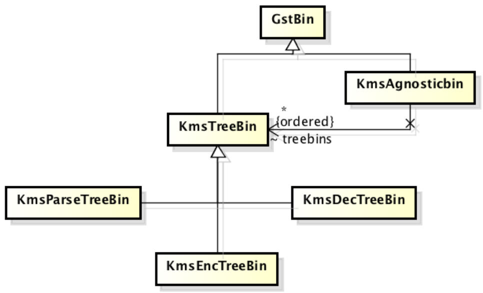

Figure 11: KmsAgnosticBin UML diagram

From Figure 11, it follows that KmsAgnosticBin is a GStreamer element. It allows connecting elements with compatible media, but with different capabilities (codec, dimensions, bitrates, etc.), or *caps* for short Moreover, it distributes the media, in different formats, to several sinks, allowing sending video to elements of different types. For this purpose, caps from the incoming media are compared against the ones required by the connected element, decoding the incoming stream as needed. Media is then re-encoded as needed, in order to comply with the caps requirements from the source pad. Provided an element may have more than one source pad, and taking into account encoding is a costly process, the agnostic bin aims to reuse the same reencoded stream if possible.

The internal structure of the agnostic bin is depicted in Figure 12.

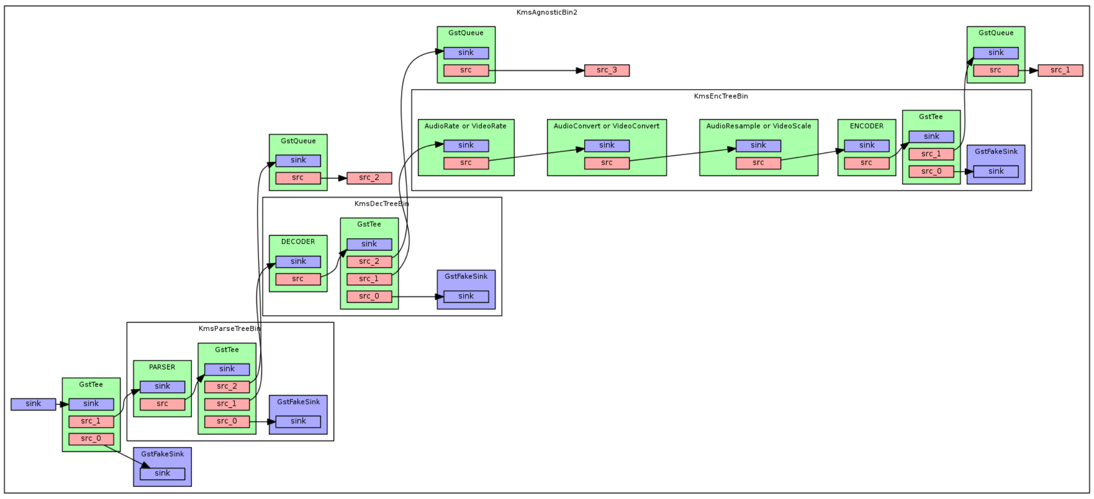

Figure 12: KmsAgnosticBin internal structure

6.1.3 Hubs

Hubs are the family of elements enabling group communications. Each MediaElement that connects to a hub will do so through a special MediaElement called HubPort, which can be connected and disconnected from Hubs to inject or receive media.

The base class for all hubs is the abstract class BaseHub, and provides extending classes with the ability to connect and disconnect HubPorts for media transmission. The following diagram shows the inheritance model, and the different types of hubs that exist. It is clear from, that hubs are another type of MediaElement, that are not KmsElements.

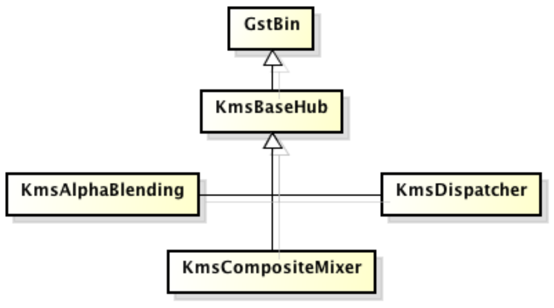

Figure 13: KmsBaseHub UML diagram

As stated before, Hubs send and receive media through HubPorts, a special type of KmsElement. This implies that, in order to send/receive media from/to a hub element, it will have to first be connected to a HubPort, and the port in turn will have to be connected to the hub itself.

6.1.3.1 Composite

This element allows combining several input streams in one output stream, divided in two columns, and as many rows as needed depending on the incoming streams. Thus, if there are eight input streams, the output stream will have two columns and four rows. All videos will have the same size in the grid, even if their size is different. The streams will appear in the grid from left to right, top to bottom in order of connection.

One of the most appealing uses of this hub, is to save bandwidth in a multiconference scenario. Without it, if 10 users are to hold a meeting, each user would be emitting it's own media, and receiving the media from the other 9 participants. Having a total of 10 connections per user, this amounts to 100 connections in the media server. Apart from the amount of bandwidth consumed in the server, these numbers are more important in the world of mobile communications, where portable devices connected through 3G/4G networks, depend heavily on the quality of their link, and most of the times have monthly quotas for internet traffic. Thus, combining all streams into one single feed, each user can have only one incoming media feed, having a total of only 2 connections per user. This does not only save bandwidth, but in some cases also computing power, since some transports encrypt media (WebRTC, for instance), which is a power-hungry operation. So by reducing the amount of connections, we can also reduce the computational power required in the client, thus allowing less powerful devices to participate in such multiconference applications.

The following figure represents the internal structure of a Composite element, with it's internal GStreamer elements shown in the KmsCompositeElement box. The grey boxes outside of this box, represent the audio and video sinks of 5 different HubPorts, and appear in the schema in order to conceptually show how the video goes from the source pads in the KmsCompositeMixer, to the sink pads in the HubPort.

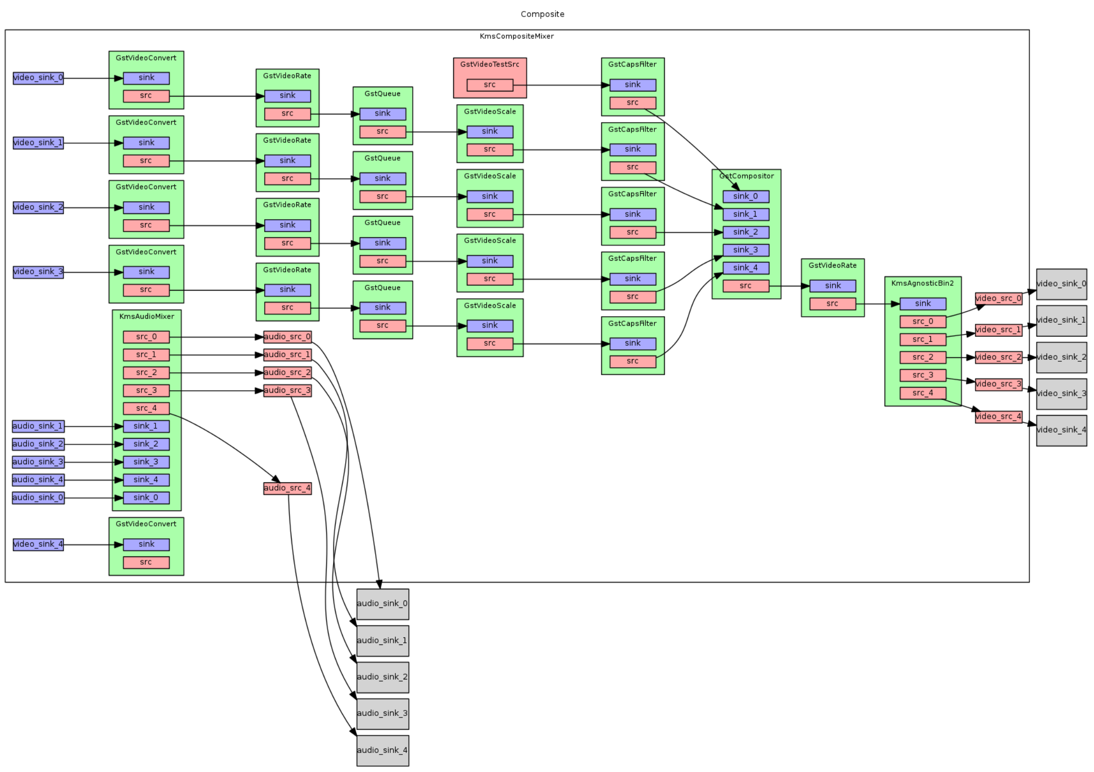
Figure 14: Composite internal structure

6.1.3.2 Dispatcher

The dispatcher is a hub that allows selecting which media stream will receive each of the clients connected to it.

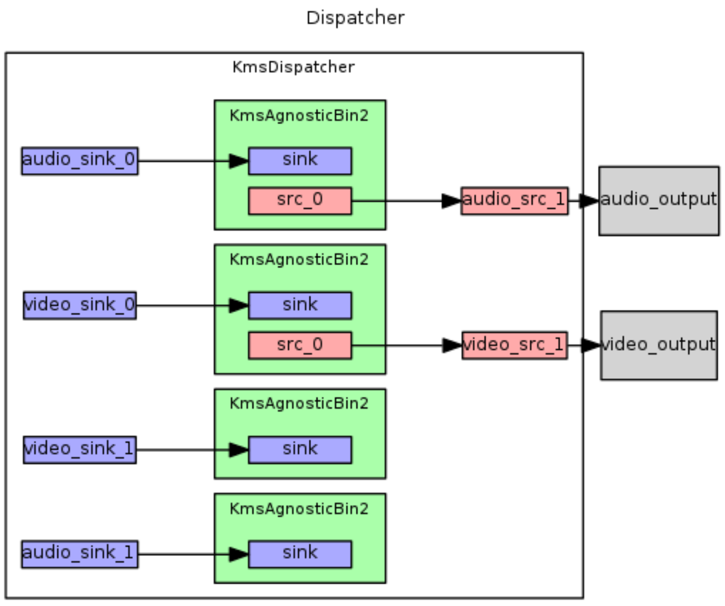

Figure 15: Dispatcher internal structure

6.1.3.3 Alpha blending

This type of hub allows showing several input streams over a master stream. This master stream defines the output size, and the other streams should adapt, according to their configuration, to the master. The internal structure is represented in the following picture. As in the Composite, the grey boxes outside of the KmsAlphaBlending box represent different HubPorts connected to it.

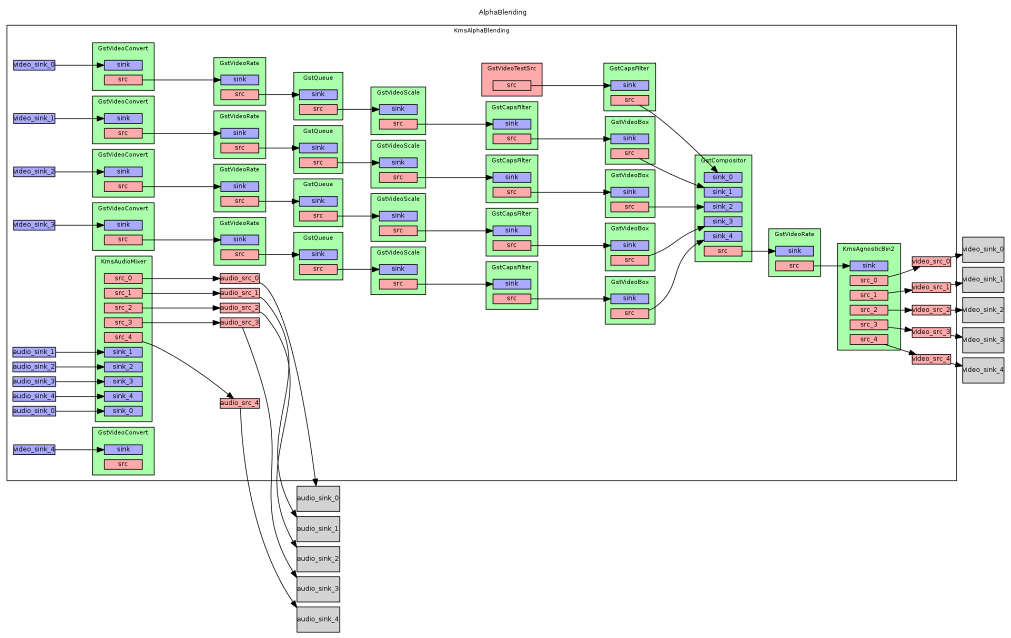

Figure 16: Alpha Blending internal structure

All streams not acting as master can be configured in their size and position (relative to the master), defining where the stream should be shown. The Z coordinate can also be modified, in order to offer several presentation layers.

6.1.4 Kurento Media Server Modules

For the objectives of developers, the most important part of the KMS architecture is its pluggable module mechanism. KMS modules come in three flavors: main modules, built-in modules and custom modules.

- Main modules. They are integrated inside KMS so that KMS cannot be deployed without the presence of them. They are organized in three different groups
  - kms-core: These are the main components of KMS comprising abstract base classes that are later extended by specific modules.
    - MediaObject: Base for all objects that can be created in the media server.
    - ServerManager: This is a standalone object for managing the MediaServer.
    - SessionEndpoint: Session based endpoint. A session is considered to be started when the media exchange starts. On the other hand, sessions terminate when a timeout, defined by the developer, takes place after the connection is lost.
    - Hub: A Hub, as described in Hubs, is a routing MediaObject. It connects several endpoints together.
    - Filter: Base interface for all filters. This is a certain type of MediaElement, that processes media injected through its sinks, and delivers the outcome through its sources.
    - Endpoint: Base interface for all end points. An Endpoint is a MediaElement that allows KMS to interchange media contents with external systems, supporting different transport protocols and mechanisms such as RTP, WebRTC, HTTP, file, URLs... An Endpoint may contain

both sources and sinks for different media types, to provide bidirectional communication.

    - HubPort: This MediaElement specifies a connection with a Hub.
    - UriEndpoint: Interface for endpoints that require a URI to work. An example of this would be a PlayerEndpoint whose URI property could be used to locate a file to stream.
    - MediaPipeline: A pipeline is a container for a collection of MediaElements and MediaMixers. It offers the methods needed to control the creation and connection of elements inside a certain pipeline.
    - SdpEndpoint: Implements a SDP negotiation endpoint able to generate and process offers/responses and configures resources according to negotiated Session Description
    - BaseRtpEndpoint: Base class to manage common RTP features.
    - MediaElement: Basic building blocks of the media server, that can be interconnected through the API. A MediaElement is a module that encapsulates a specific media capability. They can be connected to create media pipelines where those capabilities are applied, in sequence, to the stream going through the pipeline. MediaElement objects are classified by its supported media type (audio, video, etc.)
  - kms-elements: These are implementations of the specific built-in media elements provided by KMS off the self. In particular, all KMS endpoints (i.e. all media elements providing the capability of getting media into or out of a media pipeline are in these group) and also all KMS hubs (devoted to group communications and mixing). Among them we can find
    - WebRtcEndpoint: An endpoint providing full-dupplex WebRTC communication capabilities. Details on this endpoint are provided below on this document.
    - RtpEndpoint: An endpoint providing full-duplex RTP communication capabilities.
    - PlayerEndpoint: An endpoint capable of reading files from URIs and feeding the associated media into the pipeline. Currently, HTTP, RTSP and FILE URIs are supported for containers based on WebM and MP4 formats.
    - RecorderEndpoint: And endpoint capable of writing files to URIs providing the media in WebM or MP4 formats. Currently HTTP URIs (through PUT method) and FILE URIs are supported.
    - HttpGetEndpoint: An endpoint capable of providing media through pseudo-streaming in a compatible way with the HTML5 video tag. Currently only WebM format is supported.
    - HttpPostEndpoint: End endpoint capable of receiving media through pseudo-streaming compatible with HTTP Post multipart mechanism (i.e. HTML5 forms).
  - kms-filters: This contains implementations of some filters (i.e. media processing elements) which are distributed off the self with KMS. In particular, this group contains the following:
    - FaceOverlayFilter: A filter replacing faces detected into the video stream with an overlaid image. This filter is provided for testing and validation purposes.

        - ZBarFilter: A filter capable of reading bar codes and QR codes and publishing the information as an event. This filter is provided for testing and validation purposes.
        - GStreamerFilter: it is an element that allows using any GStreamer element that inherits from GstVideoFilter. Any filter property can be modified during the construction of this element, but not once it has been created. The command parameter indicates the element to be used, and defines its properties. It is used as it would be issued in gst-launch.
- Built-in modules. These are extra modules developed by the Kurento team to enhance basic KMS capabilities. These modules are not distributed off the shelf with KMS, and require being installed by the end-user to be accessible. Currently, the built-in modules are the following:
    - kms-pointerdetector: This filter detects pointer in a video stream based on color tracking.
    - kms-chroma: This filter makes transparent a color range in the top layer, revealing another image behind.
    - kms-chrowddetector: This filter detects people agglomeration in video streams.
    - kms-platedetector: This filter detects vehicle plates in video streams.
- Custom modules: These are extensions to KMS created by third parties. Hence, developers external to the Kurento team always create custom modules. In particular, filters created in the context of NUBOMEDIA deliverables D4.4.1 and D4.5.1 are considered as custom modules.

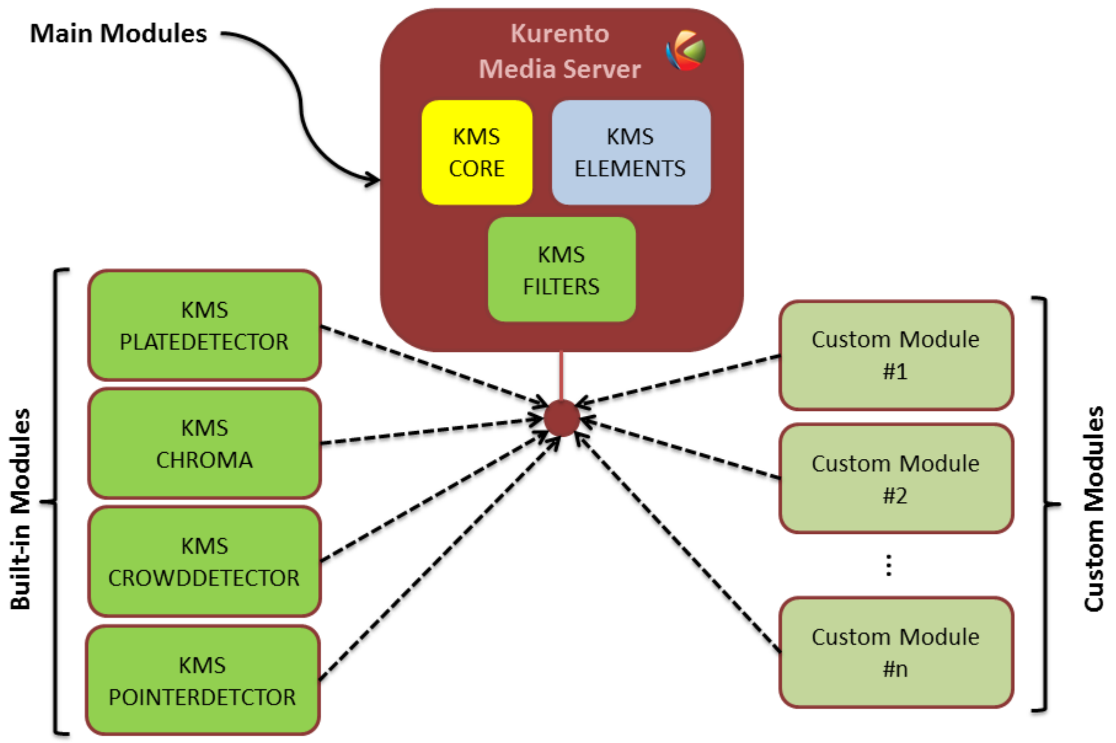

Figure 17: Kurento modules architecture is based on three different type of modules: core modules, which are distributed off the self with Kurento, built-in modules, which need to be downloaded and installed independently but are maintained by the Kurento team, and custom modules, which are created and maintained by third party organizations.

6.1.5 Distributed garbage collector

The Distributed Garbage Collector (DGC for short) is in charge of collecting media objects that are no longer in use. Objects are associated with sessions, allowing several different sessions to reference the same object.

The DGC periodically checks all active sessions, with an interval of 120 seconds, and sessions are considered inactive after two consecutive cycles without activity: keepalive has been sent or a method from an internal object has been invoked. Thus, if a session has been idle for 240 seconds, the session is destroyed, and the association between the object and the session disappears. All objects that are not referenced by any other session are destroyed along with the expiring session. When a connection-oriented protocol is used (i.e. websockets), the transport automatically keeps the session alive, and injects the session ID in the requests.

Objects can be associated/dissociated from a session with the `ref` and `unref` methods. If after invoking `unref` over an object, the object is not referenced by any session, the object will be destroyed. When an object is forcefully destroyed invoking the `release` method, the operation is performed regardless of the sessions that reference it.

6.1.6 Kurento WebRtc and RTP Endpoints

These types of endpoints are KmsElements that can send or receive media using the WebRTC or RTP protocols. Since both share some features, like SDP negotiation for instance, a convenient inheritance tree shown in Figure 18 has been established.

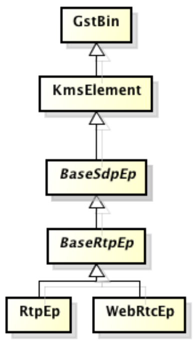

Figure 18: WebRTC and RTP endpoints UML diagram

The following sections will elaborate into the most relevant features provided by each level of the tree.

6.1.6.1 BaseSdpEndpoint

This class manages media sessions negotiated through the Session Description Protocol (SDP), which is a format for describing streaming media initialization parameters. SDP does not deliver media itself but is used by end points for negotiation of media type, format, and all associated properties. The set of properties and parameters are often called a session profile. SDP is designed to be extensible to support new media types and formats.

As stated in the previous paragraph, it is a transport and media agnostic protocol, just used during the negotiation previous to the media exchange itself. It is the responsibility of the extending classes to fulfill the exchange "contract" established in this negotiation. Since it offers a common API for generating and processing offers and answers, as per the RFC4566, these negotiation capabilities are available to child elements.

The negotiation process involves generating and SDP offer, which is in fact a declaration of its own capabilities: where can the media be sent made available (ip and port), which formats and codecs can the sender use… and is sent to the other party for analysis. The receiver the reviews all possibilities, and chooses the best possible options according to its own capabilities. With the final result, composes an SDP answer that is sent back to the original sender. This SDP answer is in fact the “how and where” of the media exchange.

6.1.6.2 BaseRtpEndpoint

This extending class encapsulates media in Real-Time Transport Protocol (RTP), which defines a standardized packet format for delivering audio and video over IP networks. This protocol is used in conjunction with the RTP Control Protocol (RTCP). While RTP carries the media streams, RTCP is used to monitor transmission statistics and quality of service (QoS) and aids synchronization of multiple streams.

RTP introduces the concept of session, which consists of an IP address and a pair of ports for RTP and RTCP for each multimedia stream. The session is managed through the GstRtpBin GStreamer element, which allows to synchronize audio and video, jitterbuffer control, etc. In the following GStreamer diagram, the GstRtpBin can be seen inside a KmsWebrtcEndpoint element. The RTP and RTCP video and audio streams are sent separately to the bin. The different green boxes inside the bin, represent the elements that manage audio and video sessions, jitterbuffer, synchronization elements… Once the media streams go through the bin, they are returned to the enclosing endpoint, which makes them available through its own source pads.

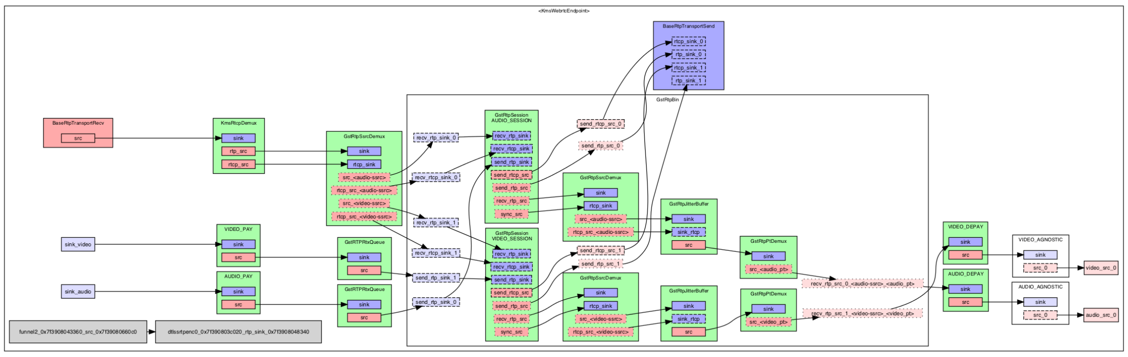

Figure 19: BaseRtpEndpoint internal structure

The RTP endpoint is transport agnostic, delegating this to extending classes. Its implementation is based on standards and draft RFCs, in order to contain better quality and interoperability with 3rd parties (as with the WebRTC stack from Chrome, for instance).

One of the most relevant features offered to extending classes, apart of course of the ability to send and receive media in real time, is to do so with the best quality possible in the present circumstances, specially referring to bandwidth. Thus, if a peer is trying to send video of a higher quality than what it’s able to transmit (for instance, trying to send FullHD video over a 3G connection) in real time, the connection will become overflowed, packages will be lost and the other peer could eventually stop receiving video at all. For this reason, when media is being transmitted with losses, the receiver will inform the sender, so the quality is lowered until it is as good as the link allows. This is called congestion control, and ensures that the clients don’t overflow the network interface, avoiding a congestive collapse taking resource reducing steps.

6.1.6.3 RtpEndpoint

This endpoint manages RTP connections using UDP as transport. As shown in Figure 18 and exposed in the previous sections, all of the capabilities are provided by the two base classes BaseRtpEndpoint and BaseSdpEndpoint, and this class is only responsible for the transport. The following image shows its internal structure where the GstRtpBin can again be seen, for providing session control, jitterbuffer and so on.

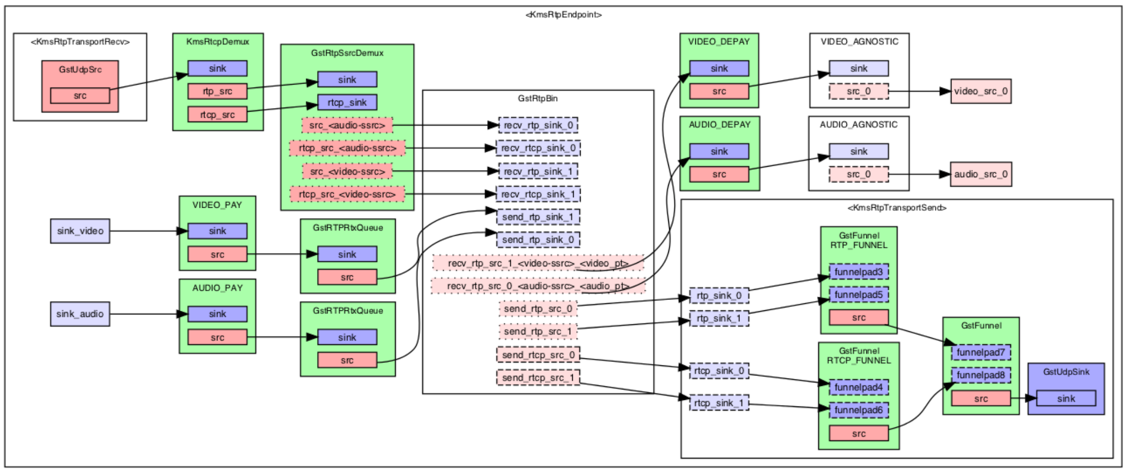

Figure 20: RtpEndpoint internal structure

6.1.6.4 WebRtcEndpoint

This endpoint creates and manages encrypted connections for media transmission, using Secure RTP (SRTP), a profile of RTP intended to provide encryption, message authentication and integrity, and replay protection to the RTP data in both unicast and multicast applications. A media security solution is created by combing this secured RTP profile and DTLS-SRTP keying. Two GStreamer elements have been purposefully created for this: GstDtlsSrtpDec y GstDtlsSrtpEnc. These two elements are in charge of the encryption and decryption of RTP packets. The can be seen in the internal schema of KmsWebRtcEndpoint shown in Figure 21, along with all the internal GStreamer elements from the endpoint. The box at the left represents the decoding element, and is connected with the sink pad of the endpoint, while the encoding element is located on the left-most side of the image, connected to the source pad of the element. For the generation of the keys, a certificate is needed, that can be provided to the WebRtcEndpoint, though it has the capability of generating the certificate, if none is provided.

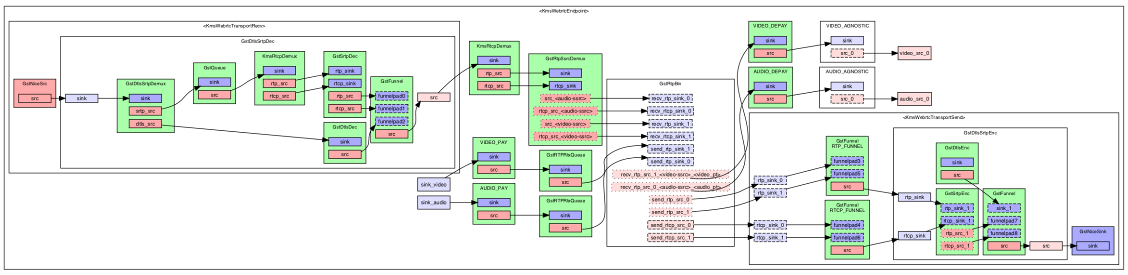

Figure 21: WebRtcEndpoint internal structure

SRTP packets are exchanged through an Interactive Connectivity Establishment (ICE) connection. ICE is a technique for NAT traversal for UDP-based media streams (though ICE can be extended to handle other transport protocols, such as TCP) established by the offer/answer model. ICE is an extension to the offer/answer model, and works by including a multiplicity of IP addresses and ports in SDP offers and answers, which are then tested for connectivity by peer-to-peer connectivity checks. The best available candidates are then chosen, and SRTP packets will be sent to those candidates, unless a better candidate is discovered during the media exchange.

For this purpose, two GStreamer-based custom-made elements are used: GstNiceSrc and GstNiceSink. The former is connected to the sink pad of the GstDtlsSrtpDec, while the later has its source connected to the sink of the GstDtlsSrtpEnc. Both *nice* elements have a common Nice Agent that manages credentials (user and password) and candidate generation.

The WebRTC standard defines a mechanism to exchange not only media, but also data. It is a feature called Data Channels, that is to be implemented in the short-medium term. This element is basically a Session Control Transmission Protocol (SCTP) connection encrypted using DTLS. In consonance with the existing structure of the WebRtcEndpoint, the approach is to create two GStreamer elements (KmsSctpDec and KmsSctpEnc) that will marshall and unmarshall SCTP packets. These two elements will be then connected to the existing GstDtlsSrtpDec and GstDtlsSrtpEnc, where the packets will be encrypted or decrypted.

6.1.6.5 *Implemented standards and drafts*

Standards:

- RFC 3711, The Secure Real-time Transport Protocol (SRTP)
- RFC 3550, RTP: A Transport Protocol for Real-Time Applications.
- RFC 4347, Datagram Transport Layer Security.
- RFC 4566, SDP: Session Description Protocol.
- RFC 4585, Extended RTP Profile for Real-time Transport Control Protocol (RTCP)-Based Feedback (RTP/AVPF).
- RFC 4588, RTP Retransmission Payload Format.
- RFC 4960, Stream Control Transmission Protocol.
- RFC 5245, Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols
- RFC 5285, A General Mechanism for RTP Header Extensions
- RFC 5764, Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)
- RFC 5761, Multiplexing RTP Data and Control Packets on a Single Port

Drafts:

- Web Real-Time Communication (WebRTC): Media Transport and Use of RTP
- Sending Multiple Media Streams in a Single RTP Session
- Negotiating Media Multiplexing Using the Session Description Protocol (SDP)
- A Google Congestion Control Algorithm for Real-Time Communication
- RTCP message for Receiver Estimated Maximum Bitrate
- The Absolute Send Time extension
- WebRTC Data Channels

6.1.7 Kurento PlayerEndpoint

This type of endpoint plays videos accessible from a URI (files, online resources, etc.) It is capable of playing all types of files supported by the GStreamer element Uridecodebin: *webm*, *mp4*, *avi*, *mkv*, *raw*, etc. The following image shows its inheritance tree.

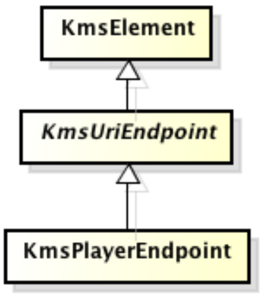

Figure 22: KmsPlayerEndpoint UML diagram

As for its internal structure,

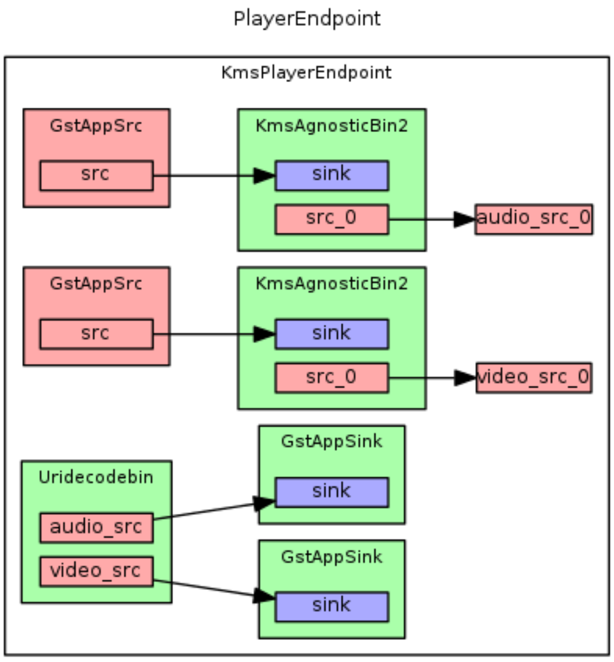

Figure 23: KmsRecorderEndpoitn internal structure

6.1.8 Kurento RecorderEndpoint

This endpoint is in charge of storing media streams in a designated location, accessible through a URI. It supports recording files in *mp4* and *webm*. The following image shows its inheritance tree.

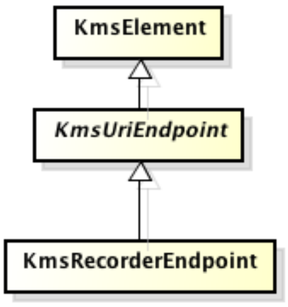

Figure 24: KmsRecorderEndpoint UML diagram

As for its internal structure,

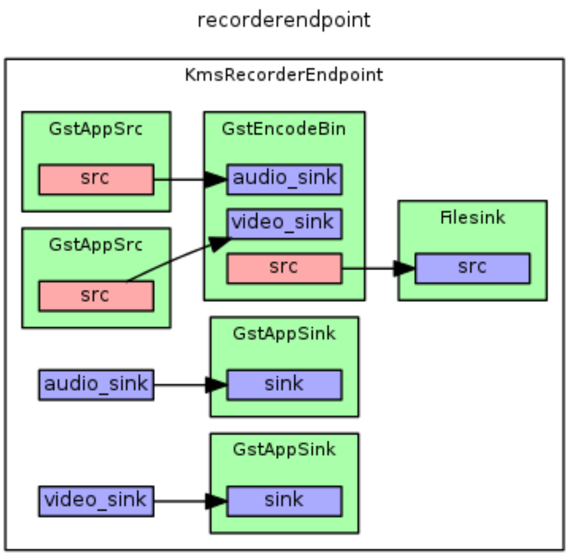

Figure 25: KmsRecorderEndpoint internal structure

6.1.9 Kurento HttpEndpoints

HttpEndpoints are elements that send or receive media through the HTTP protocol. There are two types of elements, depending of which HTTP method they use. The following figure shows the aforementioned class hierarchy.

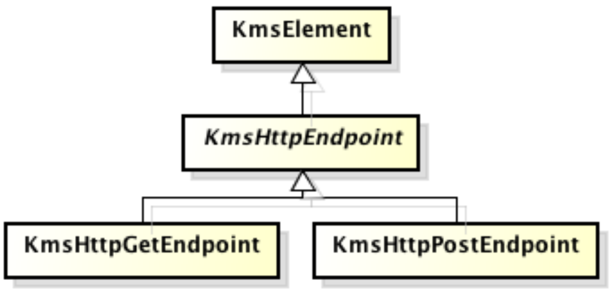

Figure 26: KmsHttpEndpoint UML diagram

6.1.9.1 HttpGetEndpoint

An HttpGetEndpoint allows publishing video in a URL, through the GET method. The video will only be accessible by one user. This is assured by requiring a cookie that is handed to the client during the first connection. The internal structure is shown in the next figure.

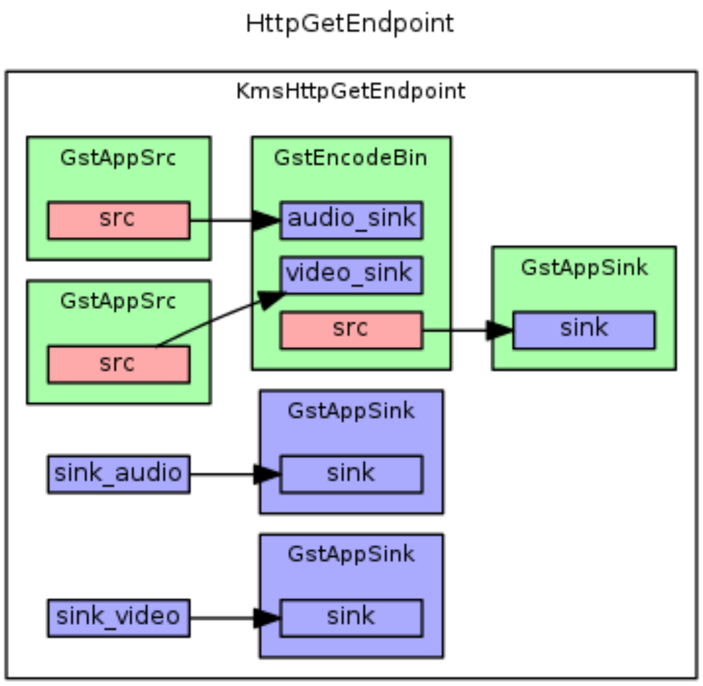

Figure 27: KmsHttpGetEndpoint internal structure

6.1.9.2 HttpPostEndpoint

This endpoint allows injecting media through the POST method. The internal structure is shown in the next figure.

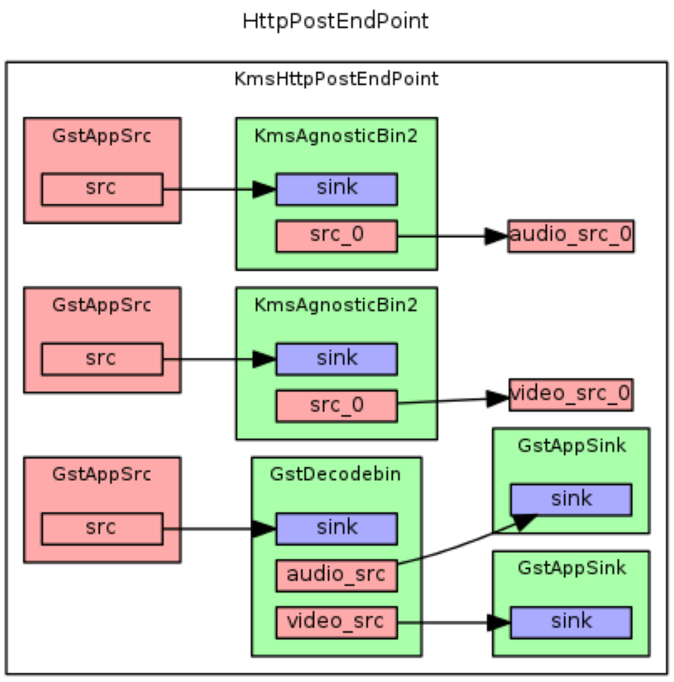

Figure 28: KmsHttpPostEndpoint internal structure

7 Implementation status (NUBOMEDIA Release 3)

7.1 Licensing models and restrictions

All software artifacts belonging to Kurento Media Server described in this document have been released and made public through the GNU Lesser General Public License (LGPL) v2.1.

Kurento Media Server extends and uses many parts of the GStreamer software stack. All these extensions and modifications have been shared with the GStreamer community in compliance with the GStreamer open source software license: LGPL v2.1.

The distribution of Kurento Media Server does not involve the distribution of GStreamer nor any of its plugins. In particular, Kurento Media Server software bundles don't integrate any kind of codec implementation. The installation of Kurento Media Server may require users to obtain and install, at their own risk, different codecs which may be protected by royalties and patents.

7.2 Obtaining the source code

All source code belonging to the Kurento Media Server described in this document can be obtained in the Kurento Github repository accessible at the following URL:

- https://github.com/kurento

Inside that repository, the following projects can be located:

- kurento-media-server
    - Description: Kurento Media Server executable, which links all KMS dependencies and orchestrate their execution as computing process.
    - Source: https://github.com/Kurento/kurento-media-server
- kms-core
    - Description: Implementation of the abstract core modules, as described above in this document.
    - Source: https://github.com/Kurento/kms-core
- kms-elements
    - Description: Implementation of the element modules, as described above in this document.
    - Source: https://github.com/Kurento/kms-elements
- kms-filters
    - Description: Implementation of the specific filter modules, as described above in this document.
    - Source: https://github.com/Kurento/kms-filters
- kms-jsonrpc
    - Description: JSON RPC connectivity providing the Kurento Protocol
    - Source: https://github.com/Kurento/kms-jsonrpc
- kms-gst-marshall
    - Description: Library for marshaling gstreamer structures required for the implementation of distributed media pipelines.
    - Source: https://github.com/Kurento/kms-gst-marshall
- kms-plugin-sample
    - Description: Example of custom module used for tutorial purposes.
    - Source: https://github.com/Kurento/kms-plugin-sample

- kms-cmake-utils
  - Description: Different utils required for the generation of custom modules
  - Source: https://github.com/Kurento/kms-cmake-utils
- kurento-module-creator
  - Description: Source code generator for client stubs of custom modules in the different languages supported by Kurento Clients.
  - Source: https://github.com/Kurento/kurento-module-creator

7.3 Obtaining the documentation

Kurento Media Server documentation has been created in Sphinx, a powerful documentation facility which transforms text documents written in the reStructuredText format into hyper-linked documentation in several formats, which include HTML, PDF, epub and LaTeX.

Kurento Media Server documentation has been made public in the following project located inside the Kurento Github repository.

- doc-kurento
  - Description: Public Kurento.org documentation
  - Source: https://github.com/Kurento/doc-kurento
  - URL: Documentation is public in http://www.kurento.org/docs/current

7.4 Installation and configuration guide

7.4.1 Installing Kurento Media Server

Kurento Media Server has to be installed on Ubuntu 14.04 LTS (32 or 64 bits).

In order to install the latest stable Kurento Media Server version 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:

```
sudo add-apt-repository ppa:kurento/kurento
wget -O - http://ubuntu.kurento.org/kurento.gpg.key | sudo apt-key add -
sudo apt-get update
sudo apt-get install kurento-media-server
```

Now, Kurento Media Server has been installed and started. Use the following commands to start and stop it respectively:

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

Kurento Media Server has a log file located at

```
/var/log/kurento-media-server/media-server.log
```

7.4.2 Kurento Media Server Configuration

The KMS configuration file is located in `/etc/kurento/kurento.conf.json`. After a fresh installation this file has the following contents:

```
{
  "mediaServer" : {
    "net" : {
      // Uncomment just one of them
      /*
      "rabbitmq": {
        "address" : "127.0.0.1",
        "port" : 5672,
        "username" : "guest",
        "password" : "guest",
        "vhost" : "/"
      }
      */
      "websocket": {
        "port": 8888,
        //"secure": {
        //  "port": 8433,
        //  "certificate": "defaultCertificate.pem",
        //  "password": ""
        //},
        "path": "kurento",
        "threads": 10
      }
    }
  },
  "modules": {
    "kurento": {
      "SdpEndpoint" : {
        "sdpPattern" : "sdp_pattern.txt"
      },
      "HttpEndpoint" : {
        // "serverAddress" : "localhost",
        /*
          Announced IP Addess may be helpful under situations such as the
server needs to provide URLs to clients whose host name is different from the
one the server is listening in. If this option is not provided, http server
will try to look for any available address in your system.
        */
        // "announcedAddress" : "localhost"
      },
      "WebRtcEndpoint" : {
        // "stunServerAddress" : "stun ip address",
        // "stunServerPort" : 3478,
        // turnURL gives the necessary info to configure TURN for WebRTC.
        //    'address' must be an IP (not a domain).
        //    'transport' is optional (UDP by default).
        // "turnURL" : "user:password@address:port(?transport=[udp|tcp|tls])",
        // "pemCertificate" : "file"
      },
      "PlumberEndpoint" : {
        // "bindAddress" : "localhost",
        /*
          Announced IP Address may be helpful under situations such as the
endpoint needs to provide an IP address to clients whose host name is
different from the one that the element is listening in. If this option is not
provided, the bindAddress will be used instead.
        */
        // "announcedAddress" : "localhost"
      }
    }
    //"module1": { …. }
    //"module2": { …. }
  }
}
```

7.4.3 Installing Kurento Media Server behind a NAT

KMS can be installed on a private network behind a router with *NAT*. The picture below shows the typical scenario.

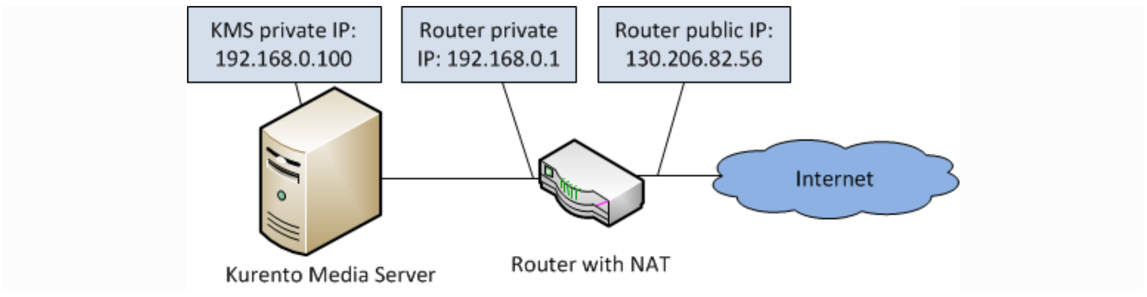

Figure 29: Typical scenario of Kurento Media Server behind a NAT

In this case, KMS should announce the router public IP in order to be reachable from the outside. In the example, sections `HttpEndpoint` and `PlumberEndpoint` within `/etc/kurento/kurento.conf.json` should be configured as follows:

```
{
  "mediaServer" : {
    "net" : {
      // Uncomment just one of them
      /*
      "rabbitmq": {
        "address" : "127.0.0.1",
        "port" : 5672,
        "username" : "guest",
        "password" : "guest",
        "vhost" : "/"
      }
      */
      "websocket": {
        "port": 8888,
        //"secure": {
        //  "port": 8433,
        //  "certificate": "defaultCertificate.pem",
        //  "password": ""
        //},
        "path": "kurento",
        "threads": 10
      }
    }
  },
  "modules": {
    "kurento": {
      "SdpEndpoint" : {
        "sdpPattern" : "sdp_pattern.txt"
      },
      "HttpEndpoint" : {
        // "serverAddress" : "localhost",
        /*
          Announced IP Addess may be helpful under situations such as the
server needs to provide URLs to clients whose host name is different from the
one the server is listening in. If this option is not provided, http server
will try to look for any available address in your system.
        */
        "announcedAddress" : "130.206.82.56"
      },
      "WebRtcEndpoint" : {
        // "stunServerAddress" : "stun ip address",
        // "stunServerPort" : 3478,
        // turnURL gives the necessary info to configure TURN for WebRTC.
        //    'address' must be an IP (not a domain).
        //    'transport' is optional (UDP by default).
        // "turnURL" : "user:password@address:port(?transport=[udp|tcp|tls])",
        // "pemCertificate" : "file"
      },
      "PlumberEndpoint" : {
        // "bindAddress" : "localhost",
        /*
```

```
          Announced IP Address may be helpful under situations such as the
endpoint needs to provide an IP address to clients whose host name is
different from the one that the element is listening in. If this option is not
provided, the bindAddress will be used instead.
        */
        "announcedAddress" : "130.206.82.56"
      }
    }
    //"module1": { …. }
    //"module2": { …. }
  }
}
```

7.4.3.1 Verifying Kurento Media Server installation

7.4.3.1.1 Kurento Media Server Process

To verify that KMS is up and running use the command:

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

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

```
nobody    1270     1  0 08:52 ?        00:01:00 /usr/bin/kurento-media-server
```

7.4.3.1.2 WebSocket Port

Unless configured otherwise, KMS will open the port **8888** to receive requests and send responses to/from by means of the *Kurento Protocol*. 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
```

7.4.3.2 Kurento Media Server Log

KMS has a log file located at `/var/log/kurento-media-server/media-server.log`. You can check it for example as follows:

```
       tail -f /var/log/kurento-media-server/media-server.log
```

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

```
[2014-10-01  10:33:23.500285]  [0x10b2f880]  [info]         KurentoMediaServer
main.cpp:239 main() Mediaserver started
```

8 References

[1] http://www.w3.org/2011/04/webrtc-charter.html
[2] https://tools.ietf.org/html/draft-ietf-avtcore-rtp-topologies-update-05

9 Annex I: Kurento.org public documentation

This annex contains Kurento.org public documentation, as it has been published in the Kurento.org web site, and corresponding to the latest stable version of the Kurento Media Server.

This documentation can be accessed on-line in the following URL

- http://www.kurento.org/documentation

Source code of this documentation (for Sphinx) can be found at the Kurento Github repository:

- https://github.com/Kurento/doc-kurento

10 Annex II: JSON-RPC protocol

The Kurento protocol provides full control of the Media Server through *Media Elements*, which are the building blocks providing a specific media functionality. They are used to send, receive, process and transform media. This API provides a toolbox of Media Elements ready to be used. It also provides the capability of creating *Media Pipelines* by joining Media Elements of the toolbox.
The API requires full-duplex communications between client and server infrastructure. For this reason, is is based on WebSocket transports.
Previous to issuing commands, the Kurento protocol client requires establishing a WebSocket connection with the server infrastructure. The URI to establish a WebSocket connection for the exchange of full-duplex JSON-RPC messages is:

```
ws://{SERVER_IP}:{SERVER_PORT}/kurento
```

Once the WebSocket has been established, the Kurento protocol offers five different types of request/response messages:

- *create*: Instantiates a new pipeline or media element in the media server.
- *invoke*: Calls a method of an existing media element.
- *subscribe*: Creates a subscription to a media event in a media element.
- *unsubscribe*: Removes an existing subscription to a media event.
- *release*: Explicit termination of a media element.

The Kurento protocol allows to servers send requests to clients:

- *onEvent*: This request is sent from server to clients when a media event occurs.

10.1 Create

Create message requests the creation of an element of the toolbox. The parameter *type* specifies the type of the object to be created. The parameter *creationParams* contains all the information needed to create the object. Each object type needs different*creationParams* to create the object. These parameters are defined later in this document. Finally, a *sessionId* parameter is included with the identifier of the current session. The value of this parameter is sent by the server to the client in each response to the client. Only the first request from client to server is allowed to not include the *sessionId* (because at this point is unknown for the client).
The following example shows a *Request* object requesting the creation of an object of the type *PlayerEndPoint* within the pipeline*6829986* and uri *http://host/app/video.mp4* in the session *c93e5bf0-4fd0-4888-9411-765ff5d89b93*:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "create",
  "params": {
    "type": "PlayerEndPoint",
    "creationParams": {
      "pipeline": "6829986",
      "uri": "http://host/app/video.mp4"
    },
    "sessionId": "c93e5bf0-4fd0-4888-9411-765ff5d89b93"
  }
}
```

The *Response* object contains the object id of the new object in the field *value*. This object id has to be used in other requests of the protocol (as we will describe later). As stated before, the *sessionId* is also returned in each response.
The following example shows a typical response to a create message:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "value": "442352747",
    "sessionId": "c93e5bf0-4fd0-4888-9411-765ff5d89b93"
  }
}
```

10.2 Invoke

Invoke message requests the invocation of an operation in the specified object. The parameter *object* indicates the id of the object in which the operation will be invoked. The parameter *operation* carries the name of the operation to be executed. Finally, the parameter*operationParams* has the parameters needed to execute the operation. The object specified has to understand the operation name and parameters. Later in this document it is described the valid operations for all object types.
The following example shows a *Request* object requesting the invocation of the operation *connect* on the object *442352747* with parameter sink *6829986*. The *sessionId* is also included as is mandatory for all requests in the session (except the first one).

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "invoke",
  "params": {
    "object": "442352747",
    "operation": "connect",
    "operationParams": {
      "sink": "6829986"
    },
    "sessionId": "c93e5bf0-4fd0-4888-9411-765ff5d89b93"
  }
}
```

The *Response* object contains the value returned while executing the operation invoked in the object or nothing if the operation doesn't return any value.
The following example shows a typical response while invoking the operation *connect* (that doesn't return anything):

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "sessionId": "c93e5bf0-4fd0-4888-9411-765ff5d89b93"
  }
}
```

10.3 Release

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

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "release",
  "params": {
    "object": "442352747",
    "sessionId": "c93e5bf0-4fd0-4888-9411-765ff5d89b93"
  }
}
```

The *Response* object only contains the *sessionID*. The following example shows the typical response of a release request:

```
{
  "jsonrpc":"2.0",
  "id": 3,
  "result":
  {
    "sessionId":"c93e5bf0-4fd0-4888-9411-765ff5d89b93"
  }
}
```

10.4 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 to the client. This kind of request is described few sections later. The following example shows a *Request* object requesting the subscription of the event type *EndOfStream* on the object *311861480*. The *sessionId* is also included.

```
{
  "jsonrpc":"2.0",
  "id": 4,
  "method":"subscribe",
  "params":{
    "object":"311861480",
    "type":"EndOfStream",
    "sessionId":"c93e5bf0-4fd0-4888-9411-765ff5d89b93"
  }
}
```

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

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

```
{
  "jsonrpc":"2.0",
  "id": 4,
  "result":
  {
    "value":"353be312-b7f1-4768-9117-5c2f5a087429",
    "sessionId":"c93e5bf0-4fd0-4888-9411-765ff5d89b93"
  }
}
```

10.5 Unsubscribe

Unsubscribe message requests the cancelation of a previous event subscription. The parameter subscription contains the *subscription* id received from the server when the subscription was created.

The following example shows a *Request* object requesting the cancelation of the subscription *353be312-b7f1-4768-9117-5c2f5a087429*.

```
{
  "jsonrpc":"2.0",
  "id": 5,
  "method":"unsubscribe",
  "params":{
    "subscription":"353be312-b7f1-4768-9117-5c2f5a087429",
    "sessionId":"c93e5bf0-4fd0-4888-9411-765ff5d89b93"
  }
}
```

The *Response* object only contains the *sessionID*. The following example shows the typical response of an unsubscription request:

```
{
  "jsonrpc":"2.0",
  "id": 5,
  "result":
  {
    "sessionId":"c93e5bf0-4fd0-4888-9411-765ff5d89b93"
  }
}
```

10.6 onEvent

When a client is subscribed to a type of events in an object, the server send an *onEvent* notification each time an event of that type is fired in the incumbent object. This is possible because the API is implemented with websockets and there is a full duplex channel between client and server. The notification that server send to client has all the information about the event:

- *data*: Information about this specific of this type of event.
- *source*: the object source of the event.
- *type*: The type of the event.
- *subscription*: subscription id for which the event is fired.

The following example shows a notification sent for server to client to notify an event of type *EndOfStream* in the object *311861480* with subscription *353be312-b7f1-4768-9117-5c2f5a087429*.

```
{
  "jsonrpc":"2.0",
  "id": 6,
  "method":"onEvent",
  "params":{
    "value":{
      "data":{
        "source":"311861480",
        "type":"EndOfStream"
      },
      "object":"311861480",
      "subscription":"353be312-b7f1-4768-9117-5c2f5a087429",
      "type":"EndOfStream",
    },
    "sessionId":"4f5255d5-5695-4e1c-aa2b-722e82db5260"
  }
}
```

The *Response* object does not contain any information. Is only a form of acknowledge message. The following example shows the typical response of an *onEvent* request:

```
{
  "jsonrpc":"2.0",
  "id":6,
  "result": ""
}
```

10.7 Error responses

If errors arise processing a request, there is a generic error response, in which an error code and a description message in sent, as follows:

```
{
  "jsonrpc": "2.0",
  "id": 7,
  "error":
  {
    "code": -32601,
    "message": "Error description"
  }
}
```

11 Annex III: Kurento Module Definition

The Kurento Module Descriptor (KMD) is a JSON used to describe modules, remote classes (with their constructor, methods, properties and events), complex types and events.

The KMD allows defining the module name, the module version, the Kurento version needed by the module and data about the API packages. Kurento version follows semantic versioning.

```
{
  "name": "yourModuleName",
  "version": "1.0.0",
  "kurentoVersion": "^5.0.0",
  "code": {
    "api": {
      "java": {
        "packageName": "yourPackageName",
        "mavenGroupId": "yourMavenGroupName",
        "mavenArtifactId":
"yourMavenArtifactName"
      },
      "js": {
        "nodeName": "moduleName",
        "npmDescription": "",
        "npmGit": "gitRepository"
      }
```

In KMD files, it is possible to use many basic types for variables as:

- `int`
- `float`
- `boolean`
- `String`
- Array of any type using `[]` (e.g.: `int[]`)

If needed, it is also possible to define extra types inside the object "`complexTypes`". For each type, the following fields must be defined:

- `name`. The type name.
- `doc`. Documentation about the new type.
- `typeFormat`. There are two different types:
  - `REGISTER`
  - `ENUM`
- `properties`. Different values for an `ENUM` type or different fields for a `REGISTER` type.
  - `name`. The property name.
  - `doc`. Documentation about the property.
  - `type`. Any basic or complex type.
  - `optional`. true indicates that the field is not mandatory.
  - `defaultValue`. Property default value.

```
"complexTypes": [
    {
      "name": "",
      "doc": "",
      "typeFormat": "",
      "properties": [
        {
          "name": "",
          "doc": "",
          "type": "",
          "optional":,
          "defaultValue":

        }
      ]
    }
  ]
```

KMD also allows defining events. An event has the following fields:

- `name`: The event name.
- `doc`: Documentation about the event.
- `extends`: Base class for the event.
- `properties`: All the event fields. These variables will be accessible from the API clients.
  - `name`: The property name.
  - `doc`: Documentation about the property.
  - `type`: Any basic or complex type.
  - `optional`: true indicates that the field is not mandatory.
  - `defaultValue`: Property default value.

```
  "events": [
    {
      "name": "EventName",
      "extends": "Media",
      "doc": "",
      "properties": [
        {
          "name": "",
          "doc": "",
          "type": "",
          "optional":,
          "defaultValue":
        }
      ]
    `
```

KMD describes the classes implemented in Kurento. For each class you can define the following:

- `name`: Class name.
- `extends`: Parent Class.
- `doc`: Documentation about the class.
- `constructor:`
    - `doc`
    - `params`
        - `name`. Parameter name.
        - `doc`
        - `type.` Any basic or complex type.
- `methods`: Define accessible methods from the API.
    - `name`: Method name.
    - `doc`
    - `params`: Method parameters.
        - `name`: Parameter name.
        - `doc`
        - `type:` Any basic or complex type.
- `properties`. Define the class properties accessible from the API.
    - `name`. Property name.
    - `doc.`
    - `type`. Any basic or complex type.
    - `final`. `true` indicates that the property value cannot change.
    - `optional`. `true` indicates that the field is not mandatory.
    - `defaultValue`. Property default value.
- `events`. Name of the event thrown by the module.

```
   "remoteClasses": [
     {
       "name":
"className",
       "extends":
"parentClass",
       "doc": "",
       "constructor": {
         "doc": "",
         "params": [
           {
             "name":
"paramName",
             "doc": "",
             "type":
"paramType"
           }
         ]
       },
       "methods": [
         {
           "name":
"methodName",
           "doc": "",
           "params": [
             {
               "name":
" paramName ",
               "doc":
```

12 Annex IV: Kurento Module Creation Procedure

Kurento is a pluggable framework. Each plugin in Kurento is called **module**. We classify Kurento modules into three groups, namely:

- Main modules. Incorporated out of the box with Kurento Media Server
  - `kms-core`: Main components of Kurento Media Server.
  - `kms-elements`: Implementation of Kurento Media Elements (`WebRtcEndpoint`, `PlayerEndpoint`…).
  - `kms-filters`: Implementation of Kurento Filters (`FaceOverlayFilter`, `ZBarFilter`, `GStreamerFilter`).
- Built-in modules. Extra modules developed by the Kurento team to enhance the basic capabilities of Kurento Media Server. So far, there are four built-in modules, namely:
  - `kms-pointerdetector`: Filter that detects pointers in video streams based on color tracking. The command to install this module is:

    ```
    sudo apt-get install kms-pointerdetector
    ```
  - `kms-chroma`: Filter that makes transparent a color range in the top layer, 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
```

- Custom modules. Extensions to Kurento Media Server which provides new media capabilities.

The following picture shows a schematic view of the Kurento Media Server as described before:

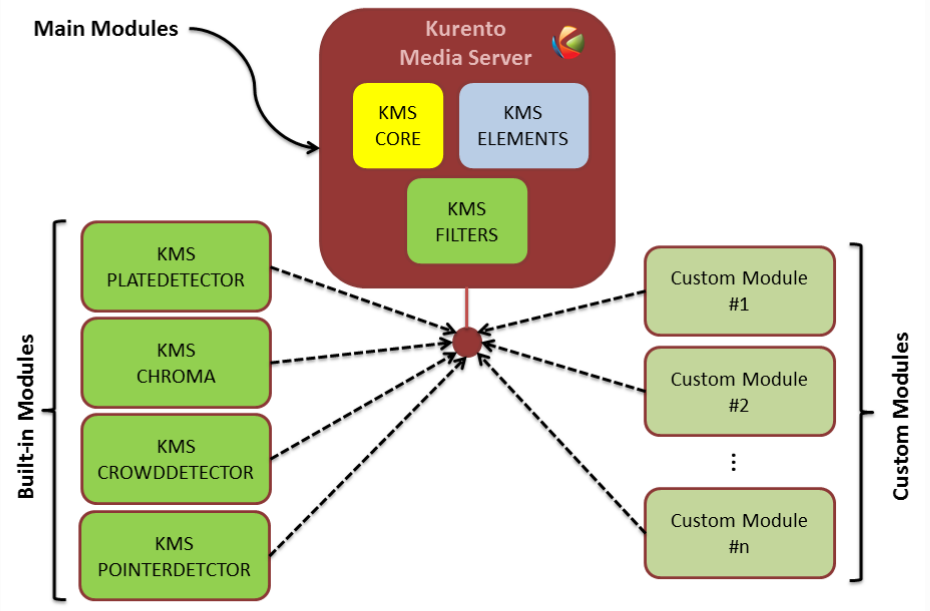

Figure 30: Kurento modules architecture. Kurento Media Server can be extended with built-in modules (crowddetector, pointerdetector, chroma, platedetector) and also with other custom modules

Taking into account the built-in modules, the Kurento toolbox is extended as follows:

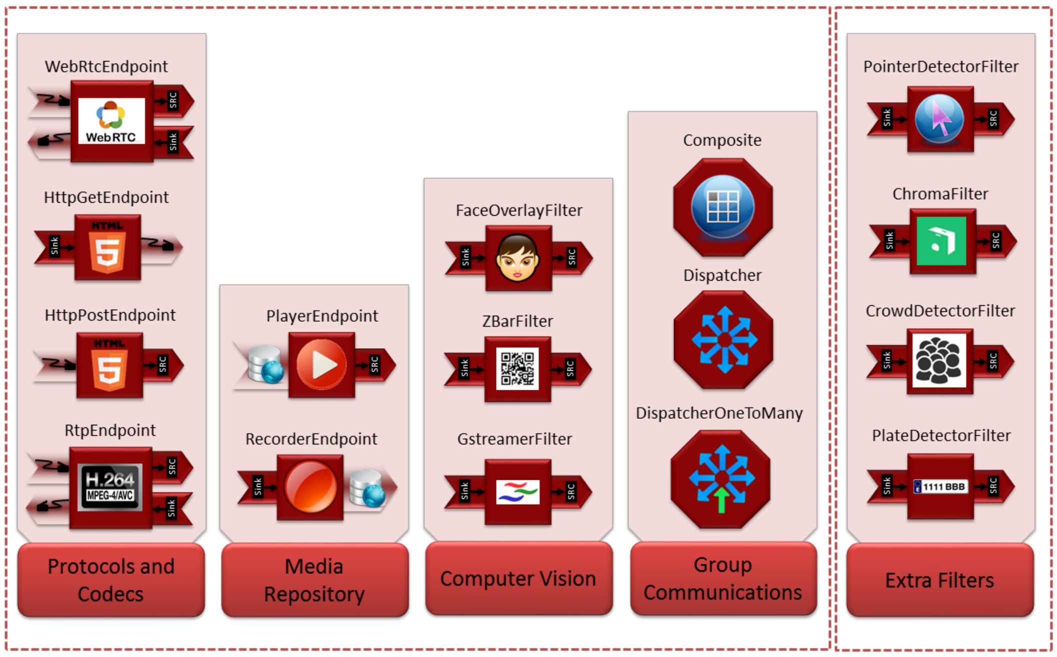

Figure 31: Extended Kurento Toolbox. The basic Kurento toolbox (left side of the picture) is extended with more computer vision and augmented reality filters (right side of the picture) provided by the built-in modules.

12.1 How to Develop Kurento Modules

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

- Modules based on *OpenCV*: These kinds of modules are recommended if you would like to develop a computer vision filter.
- Modules based on *GStreamer*: These kinds of 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 creating 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 in the shell:

```
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 `CmakeLists.txt` files necessary and example files of Kurento module descriptor files (.kmd). These files describe our modules, the constructor, the methods, the properties, the events and the complex types defined by the developer.

Once, `kmd` files are completed we can generate code. The tool `kurento-module-creator` generates glue code to server-side. From the root directory:

```
cd build
cmake ..
```

The following section details how to create your module depending on the filter type you chose (OpenCV or GStreamer):

12.2 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 this function.

12.3 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.

12.4 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 in `src/server`. The following parameters are available:

- `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 allows 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
- `SERVER_STUB_DESTINATION` (required): The generated code that you may need to modify will be generated on the folder indicated by this parameter.

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:

You can generate the Debian package (`debuild -us -uc`) and install it (`dpkg -i`). 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`
  This command 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 to execute
  `cmake .. -DGENERATE_JS_CLIENT_PROJECT=TRUE`.

This command generates a `js` folder with client code. Now you can add the JavaScript library to use your module in your application manually. Alternatively, you can use *Bower* (for JavaScript for browser) or *NPM* (for JavaScript 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"
}
```

12.5 Examples

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

- OpenCV module
- GStreamer module

There are a lot of examples of 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