D4.3	
Version	2.1
Author	NAEVATEC
Dissemination	PU
Date	30/11/2016
Status	Final

D4.3: NUBOMEDIA Media Server and modules v3

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:	Report
Lead editor:	Javier López
Planned delivery date	11/2016
Actual delivery date	30/11/2016
Keywords	Media Server, Kurento, Media Capabilities, VCA, AR

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

SEVENTH FRAMEWORK PROGRAMME Information Society Technologies European Commission

FP7 ICT-2013.1.6. Connected and Social Media

Contributors:

Javier López (NAEVATEC)
Iván Gracia (NAEVATEC)
José A. Santos Cadenas (NAEVATEC)
Miguel París Díaz (URJC)
David Fernández López (NAEVATEC)
Boni García (URJC)
Luis López (URJC)
Victor Hidalgo (VTOOLS)
Satu-Marja Mäkelä (VTT)

Internal Reviewer(s):

Luis López (URJC)

Version History

Version	Date	Authors	Comments
0.1	25-11-2015	Luis López	Initial Version
0.2	01-12-2015	Boni García	Software architecture of the media server
0.3	10-12-2015	David Fernández	Custom modules
0.4	12-12-2105	Victor Hidalgo	VCA modules
0.5	15-12-2015	Satu-Marja Mäkelä	AR modules
0.6	10-01-15	Luis Lopez	Updated figures
1.0	21-01-2015	Luis López	Integrated connectors
1.1	2-02-2015	José Antonio Santos	Revision of Media Server architecture updating for D4.3
2.0	01-09-2016	Jose Antonio Santos	Completed information about multi-sensory/multi-domain capabilities
2.1	03—10-2016	Luis Lopez	Completed information about agnostic ib

Table of contents

List of Figures:

Acronyms and abbreviations:

API	Application Programming Interface
AR	Augmented Reality
FBO	Frame Buffer Object
IMS	IP Multimedia Subsystem
IoT	Internet of Things
KMS	Kurento Media Server
MCU	Multipoint Control Unit
MSM	Media Switching Mixer
RTC	Real-Time Communications
RTP	Real-time Transport Protocol
SCTP	Stream Control Transmission Protocol
SFU	Selective Forwarding Unit
VCA	Video Content Analysis
WebRTC	Web Real Time Communications

1 Executive summary

This document contains a description of the activities carried out in the context of the NUBOMEDIA project for WP4 (NUBOMEDIA Cloud Elements and Pipelines). In particular, it contains a description of the NUBOMEDIA Media Server: a real-time media infrastructure susceptible of providing to NUBOMEDIA all its low-level media capabilities which include media transport, media encoding, media decoding, media processing, Video Content Analysis, Augmented Reality and media interoperability. The NUBOMEDIA Media Server has been developed by evolving and extending the Kurento Media Server (KMS). Due to this, Kurento, KMS, NUBOMEDIA Media Server and Media Server are considered as synonyms along this document.

2 Introduction

NUBOMEDIA is an open source cloud PaaS (Platform as a Service), which makes possible, through simple APIs, to integrate RTC (Real-Time media Communications) and advanced media processing capabilities including VCA (Video Content Analysis) and AR (Augmented Reality). For this, NUBOMEDIA is based on a complex architecture that separates the cloud orchestration and control logic from the media capabilities themselves. This architecture has been specified in NUBOMEDIA Project Deliverable D2.4.2.

This document is specifically concentrated on describing the part of the NUBOMEDIA architecture in charge of providing the media capabilities to the platform, including the ability to transport media, store media, (de)code media, analyze media, augment media and process media in a generic sense.

3 Objectives

NUBOMEDIA WP4 addresses the objective of creating a media server for NUBOMEDIA. This objective can be split in the following sub-objectives:

- Objective 1: To create the appropriate interactive media capabilities for NUBOMEDIA including media sending and receiving, (de)coding, adapting, filtering and processing. To create the appropriate interfaces enabling their use into the cloud and their logic for chaining them to form media processing pipelines performing the desired media workflows for an application. This corresponds with global Objective 1.2 as specified in NUBOMEDIA DoW.
- Objective 2: To crate the appropriate connectors for granting interoperability, in particular with DCNs, TV broadcasting systems and IMS infrastructures. This corresponds with global Objective 1.3 of the project.
- Objective 3: To create the appropriate generalized multimedia as video + audio + multisensory data. This corresponds with global Objective 3 of the project.
- Objective 4: To crate the appropriate capabilities enabling group communications adapted to real social interactions. This corresponds with global Objective 4 of the project.

4 The NUBOMEDIA Media Server

4.1 Rationale

NUBOMEDIA aim is to create an elastic cloud infrastructure for real-time media. This infrastructure is built by replicating and orchestrating monolithic (i.e. non distributed)

processes each of which is an instance of a media server. This type of architecture is typical in many different types of distributed systems and requires two ingredients.

- The monolithic media server
- The cloud replication and orchestration capabilities.

This section is devoted to providing detailed information about the former. The NUBOMEDIA Media Server is hence a monolithic media server which has be ability of holding custom extension modules providing additional capabilities.

4.2 Objectives

The NUBOMEDIA Media Server has the objective of providing 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 loss 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.

4.3 Architectures: functional architecture and software architecture

The aim of this document is to provide detailed information enabling the understanding of the internal workings of the Media Server as well as its software structure and components. With this aim, we distinguish two types of architectures:

- Functional architecture (or just architecture). This refers to the internal structure of a system. In general the functional architecture shows which runtime components a system has and specified their relationships. All functional architectural diagrams in this document have been created following the conventions of the Fundamental Modeling Concepts [FMC] framework. Following FCM conventions, active components are depicted as named squared-corner boxes and passive components as named rounded-corner boxes. Components names comprise one or several upper-case terms. Interfaces are represented through named or un-named circles. For named interfaces, we accept the convention of them starting with a lower-case "i" character.
- Software architecture. This refers to a high level description of the organization of a complex software system. The software architecture is related to the software artifacts (i.e. the files containing the source code). Software architecture diagrams in this document are created following the UML framework [UML]

4.4 Media Server functional architecture

4.4.1 Media Server architecture

As shown in Figure 1, Media Servers hold a number of media capabilities, which may include media transport, media transcoding, media processing, media mixing and media archiving. These media capabilities are encapsulated into abstractions called Media Elements (ME), so that Media Elements hide the complexities of a specific media capability behind a comprehensive interface. MEs have different flavors (i.e. types) in correspondence with their different specialties (e.g. *RtpEndpoint*, *RecorderEndpoint*, *FaceDetectorFilter*, etc.)

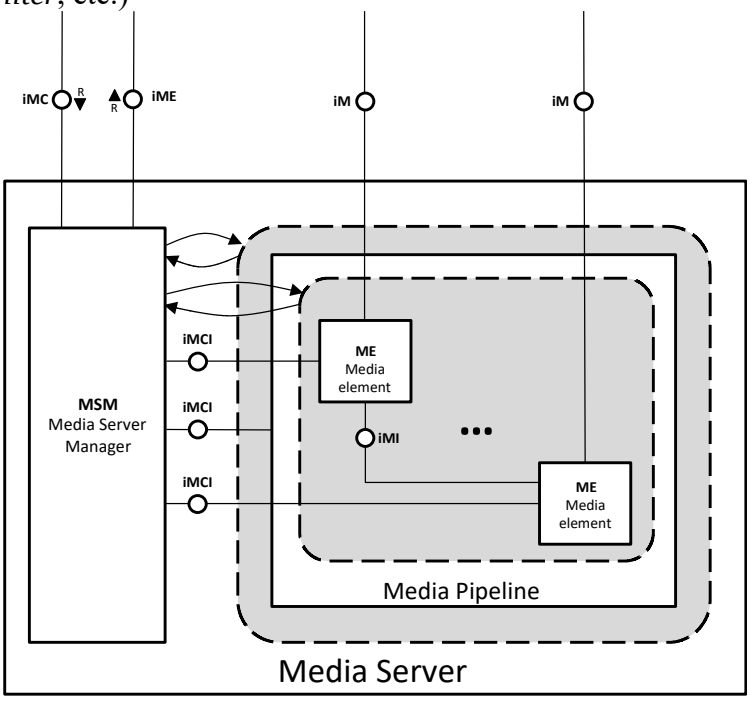

Figure 1. Media Server (MS) component internal architecture.

A MS comprises a number of Media Pipelines (MPs). MPs are graphs of interconnected Media Elements (MEs). Each ME implements a specific media capability: media transport, media archiving or media processing. MS pipelines and elements are controlled by the Media Server Manager, which

communicated with the external world through the Media Control interface (iMC) and the Media Events interface (iME).

The Media Element component

As shown on Figure 2, a Media Element can be seen as a black box taking media from a sink, applying a specific media capability (e.g. processing, transporting, etc.), and issuing media through a source. As a result, Media Element instances can exchange media among each other through the Media Internal interface (iMI) following this scheme:

- A Media Element can provide, through its source, media to many sink Media Elements.
- A Media Element can receive, through its sink, media from only one source Media Element.

Media Element instances can also exchange media with the external world through the Media interface (iM). Hence, the iM interface represents the capability of the Media Server to send and receive media. Depending on the Media Element type, the iM interface may be based on different transport protocols or mechanisms. For example, in a Media Element specialized in RTP transport, the iM interface shall be based on RTP. In a Media Element specialized in recording, the iM interface may consist on a file system access.

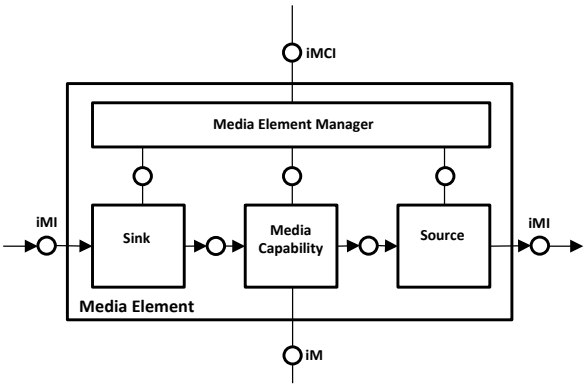

Figure 2. Media Element (ME) component internal architecture.
A ME has a sink, where media may be received and a source, where media can be send. Sinks and Sources exchange media with other MEs through the Media Internal interface (iMI). ME specific capabilities are encapsulated behind an interface that it made available to the rest of MS components through the Media Control Internal interface (iMCI)

Remark, however, that not all Media Elements need to provide a iM interface implementation. Media Elements having the ability of exchanging media with the external world (and hence, providing an iM implementation) are called **Endpoints**. Media Elements which do not provide an iM implementation are called **Filters**, because their only possible capabilities are to process and transform the media received on its sink. Their architectural differences are shown on Figure 3.

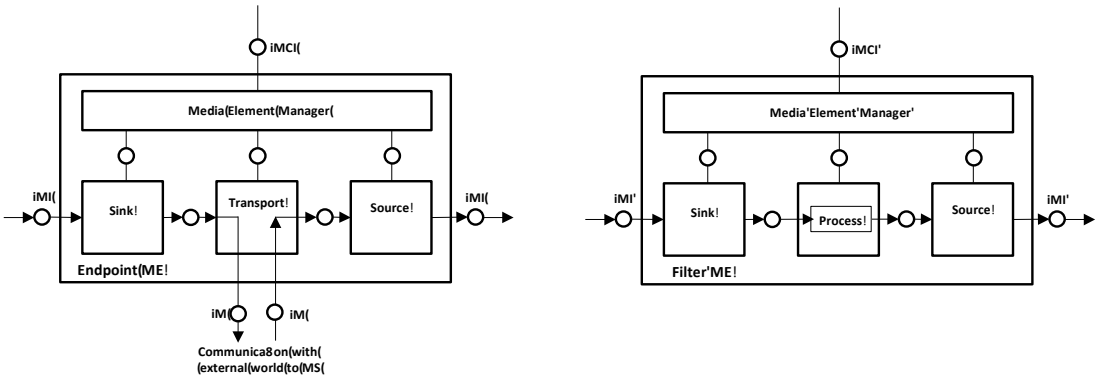

Figure 3. Media Server (MS) Filters.
Endpoint MEs (left) are MEs with the capability of exchanging media with the external world through the iM interface. In an Endpoint, media received through its Sink is forwarded to an external entity (e.g. the network). Media received from the external entity is published into the Endpoint Source. Typically, Endpoint MEs publish events associated with the transport status (e.g. media-connection-established, etc.) Filter MEs (right) do not implement the iM interface and cannot exchange media with the external world. Hence, Filter MEs can only be used for processing the media received through their Sinks, which is then published to their Sources. Filter MEs typically publish events associated to the specific media processing being performed (e.g. face-detected in a FaceDetector filter)

The Media Element behavior is controlled by the Media Element Manager (MEM), which communicates with other MS components through the Media Control Internal interface (iMCI). The iMCI provides

- A mechanism for sending commands to the Media Element.
- A mechanism for publishing media events and state information from the Media Element.

There is special type of Media Element called a Media Hubs. A Media Hub extends Media Elements capabilities with the ability of receiving a variable number of media streams at its input. In other words, Media Hubs are Media Elements with a variable number of sinks. Media Hubs are typically used for managing groups of streams in a consistent and coherent way.

The Media Pipeline component

Media Element capabilities can be complemented through a pipelining mechanism meaning that media issued by a specific Media Element can be fed into a subsequent Media Element whose output, in turn, could be fed with the next Media Element and so on and so forth. A graph of interconnected Media Elements is called a Media Pipeline (MP). Hence, a Media Pipeline is a set of Media Elements plus a specification of their interconnecting topology. Remark that Media Pipeline topologies do not need to be "chains" (sequences of interconnected Media Elements) but can take arbitrary graph topologies as long as the Media Element interconnectivity rules specified above are satisfied.

Hence, in a Media Pipeline one can combine in a very flexible way Media Element capabilities enabling applications to leverage different types of communication topologies among end-users, which may include any combination of simplex, full-duplex, one-to-one, one-to-many and many-to-many.

Media Pipeline capabilities are managed by the Media Pipeline Manager module, which controls the lifecycle of its composing Media Elements, forwards them commands from the external world and publishes Media Element events to subscribers. The Media Pipeline Manager uses the iMCI interface for performing these actions.

A Media Server instance can hold zero or more Media Pipelines.

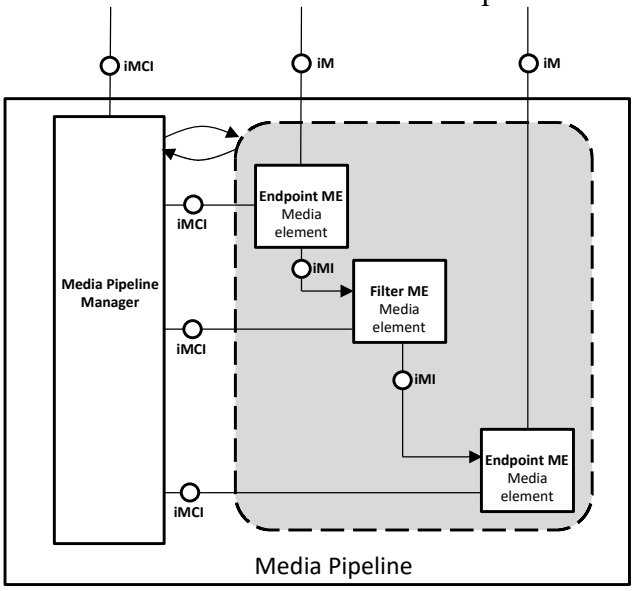

Figure 4. Media Pipelines (PMs).

Media Pipelines (PMs) are graphs or interconnected Media Elements (MEs) that implement a specific media processing for an application. The Media Pipeline Manager controls the lifecycle and behavior of its MEs by brokering control messages and events among the MEs and the Media Server components out of the MP.

The Media Server Manager component

As shown in Figure 1, the Media Server Manager (MSM) component manages the lifecycle and behavior of MPs and MEs on behalf of applications. For this, The Media Server Manager exposes two interfaces to the external world:

- The Media Control Interface (iMC): This interface is based on a request/response mechanism and enables applications to send commands to both Media Pipelines and Media Element instances. Through these commands, applications can access features such as:
  - Create and delete Media Pipelines and Media Elements
  - Execute specific built-in procedures on Media Pipelines or Media Elements
  - Configure the behavior of Media Pipelines and Media Elements
  - Subscribe to specific events on Media Pipelines and Media Elements

  The Media Server Manager translates commands received on the iMC interface into invocations to the corresponding Media Elements and Media Pipelines through the iMCI interface.
- The Media Events interface (iME): This interface is based on a request/response mechanism and enables Media Element instances to send events to applications, that are then acknowledged. These events may include semantic media information (e.g. detected a face in a *FaceDetector* filter), media transport information (e.g. media negotiation completed), error information, or any other kind of information considered relevant by the Media Element developer. For this purpose, the Media Server Manager translates any events received in

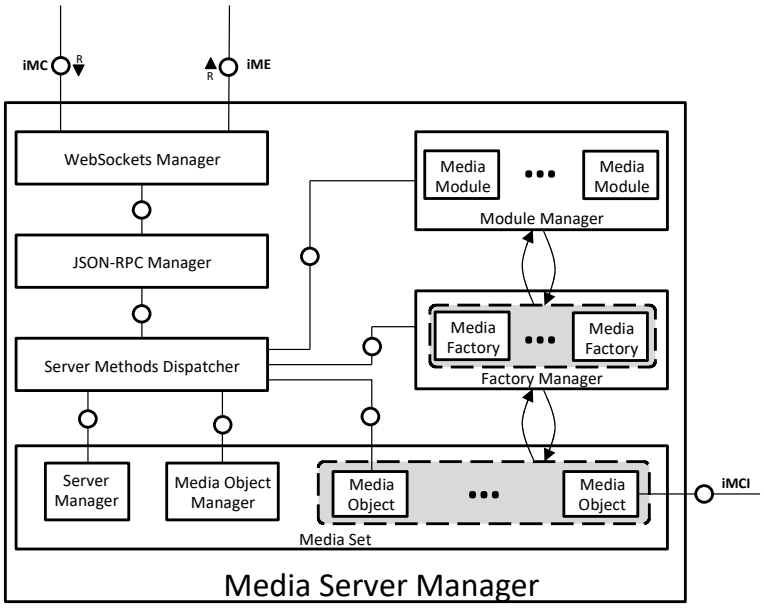

Figure 5: Media Server Manager (MSM).

The Media Server Manager (MSM) holds the appropriate logic for performing two functions. The first is to enable communications with applications through the iMC control interface and the iME event publishing interface. The second is to provide the appropriate logic for dispatching, managing and providing semantics to control messages and events, including the ones in charge of media capabilities lifecycle management (e.g. create, release, etc.)

A detailed description of the Media Server Manager internals is shown on Figure 5. As it can be observed, it comprises several modules playing specific functions each.

- The WebSockets manager provides connection-oriented full-duplex communications with applications basing on the WebSocket protocol. This module is in charge of the WebSocket line protocol and of transporting data on top of it.
- The JSON-RPC Manager provides JSON-RPC marshaling and unmarshalling of messages, accordingly to JSON-RPC v2 standard.
- The Server Methods Dispatcher (aka Dispatcher) module contains the logic for routing messages to their appropriate destinations, so that the message semantics can be provided. The Dispatcher can orchestrate actions on several other modules for obtaining that semantics.
- The Module Manager is where module information is held. A module is a collection of one or several media capabilities (i.e. Media Element). All the required information for using a module in the Media Server is called a Media Module in our architecture. Media modules are loaded upon Media Server startup. The Media Module contains the module meta information (e.g. module name, module locations, etc.) The Module Manager exposes a service for querying Media Module information to the rest of modules.
- The Factory Manager controls the lifecycle of Media Factories. Upon Media Module loading, the Module Manager registers one or several Media Factories into the Factory Manager. Each Media Factory has the ability of creating a specific media capability among the ones provided by the module. This creation process is mediated by Media Stubs, so that Media Factories only create Media Stubs and are the Media Stubs themselves which create the corresponding Media Element instances.
- The Media Set holds Media Objects, which are proxies to the media capabilities themselves (i.e. the real Media Elements and Media Pipelines). It also holds a number of facilities for managing them. In particular, the Media Object

Manager translates media object references (as managed in the iMC interface) into object references pointing to the appropriate Media Objects. The Media Objects provide a mechanism for mediating between the Media Server Manager and the media capabilities themselves. All requests coming from the iMC interface targeting a specific Media Element or Pipeline are routed by the Dispatcher to the corresponding Media Object, which provides it semantics consuming the Media Element or Pipeline primitives through the iMCI interface.

- The Media Object Manager also takes the responsibility of media session management. Media sessions are an abstraction provided by the iMC interface through a media session id. All JSON-RPC messages exchanged in the context of a session carry out the same id, which is unique for the session. The Media Object Manager leverages media sessions for implementing a Distributed Garbage Collection mechanism. For this, the Media Object Manager maintains the list of Media Objects associated to a specific media session. Whenever a media session is not having any active WebSocket connection during a given time period it is considered as a death session. In this case, all its Media Objects are released together with their corresponding media capabilities.
- The Server Manager is an object enabling Media Server introspection. For this, it queries the Media Object Manager to recover the list of Media Objects, which represent the Media Pipelines and Media Elements, as well as their interconnecting topologies. This makes possible for applications to monitor, instrument and debug what is happening inside the Media Server.

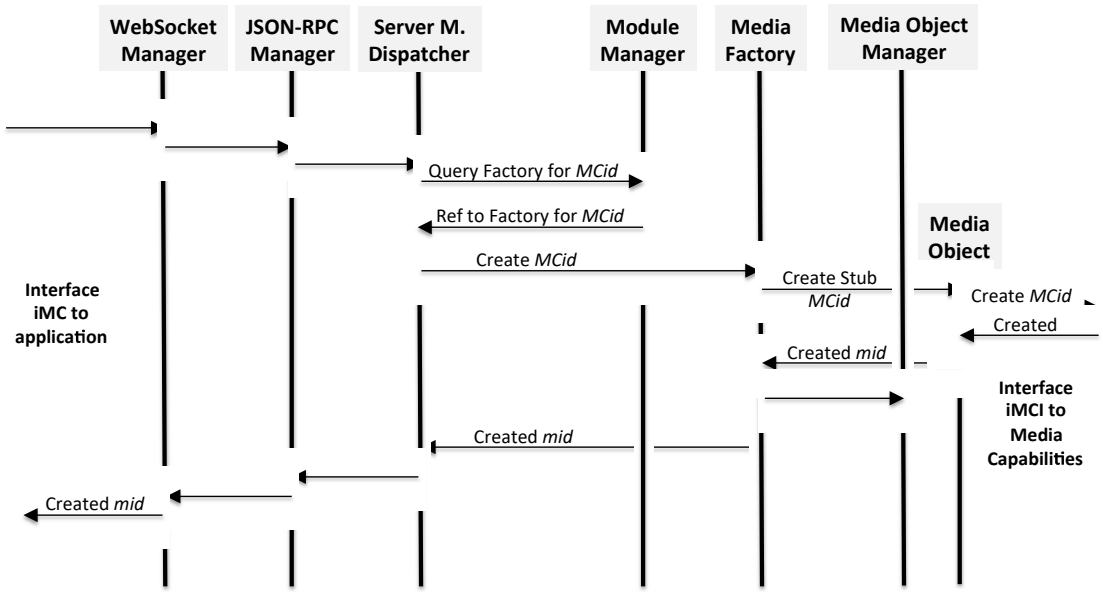

Figure 6. Media Server Manager modules interactions (I).
This flow diagram shows the interactions among the Media Server Manager modules for creating a new media capability (i.e. Media Element or Media Pipeline). The specific media capability type is identified by the string MCid (Media_Capability_id) that needs to be registered among modules capabilities into the Module Manager. Once the capability has been instantiated, it receives a unique id (mid or Media_id), which identifies it during its whole lifecycle

For understanding the interactions among MSM modules lets observe Figure 6, where the sequence diagram for the instantiation of a specific media capability is shown. From top to bottom, the steps are the following:

- A message asking for media capability creating is received at the MSM WebSocket Manager. This message needs to identify the specific media

capability type that is to be created through an ID, which we call MCid (Media_Capability_id) in the diagram.

- This message is given to the JSON-RPC Manager which unmarshalls it as a data structure.
- The unmarshalled message is given to the Dispatcher, which recognizes it as a capability creation message, and orchestrates its semantics.
- The Dispatcher queries the Module Manager for recovering the specific Media Factory knowing how to create MCid capabilities. We assume one Media Module has registered factories for MCid upon Media Server startup.
- The Module Manager provides the Dispatcher with a reference to the specific Media Factory creating MCid capabilities.
- The Dispatcher asks the Factory Manager to recover the appropriate Media Factory and commands it to create a MCid capability.
- The Media Factory Creates the appropriate Media Object associated to the MCid capability.
- The Media Object executes whatever actions are necessary for instantiating the MCid capability and for initializing it.
- The Media Factory receives the unique id of the newly created Media Object (mid).
- The Media Factory registers the new Media Object into the Media Object Manager, where it shall be recovered later when commands arrive to that specific stub.
- The mid is propagated to the application following the appropriate path (i.e. marshaling and WebSocket transport).

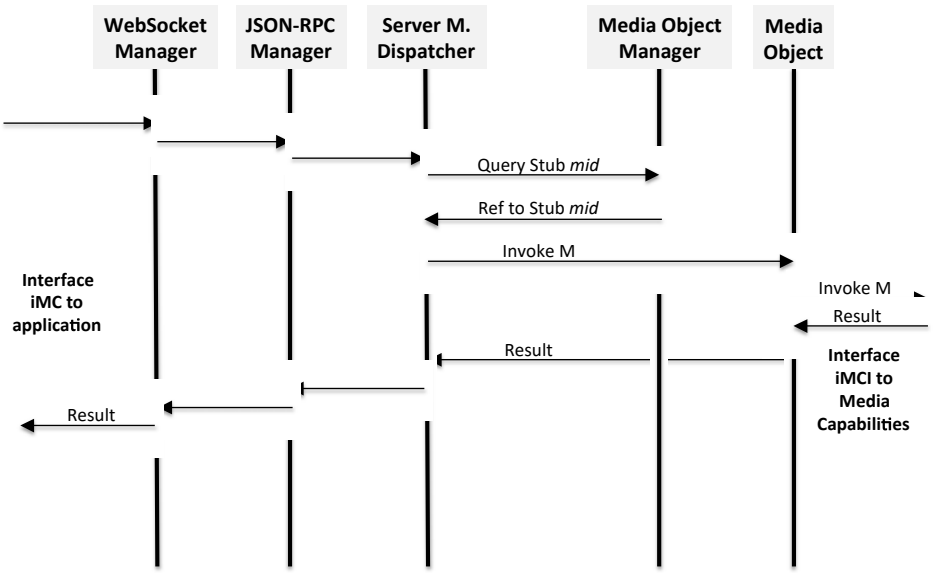

Figure 7. Media Server Manager modules interactions (II).
This flow diagram shows the interactions among the Media Server Manager modules for invoking a method (identified as M) into a specific Media Element (identified as mid)

Once a media capability (e.g. Media Element) has been created, it can be accessed and controlled following the scheme depicted in Figure 7. From top to bottom, the steps are the following:

- A request arrives from the application through interface iMC asking for invoking a specific primitive or method (identified as M in the diagram) onto the Media Element whose id is mid.
- The requests are unmarshalled at the JSON-RPC module.

- The Dispatcher detects the request has invocation semantics and orchestrates the appropriate sequence for it.
- First, the Media Object Manager is queried for recovering the Media Object whose id is mid.
- Second, the invocation is launched to the specific Media Object holding that mid. That object knows how to reach the real Media Element and executes the appropriate actions through the iMCI interface for having the primitive executed.
- As a result, a return value may be generated. This return value is dispatched back to the application following the reverse path (i.e. marshaling and WebSocket transport).

4.4.2 Media Server main interactions

As introduced above, the main interactions between the Media Plane and the external world take place through three MS interfaces: iMC and iME for control and events and iM for media exchange.

Interactions through the iMC and iME interfaces take place through a request/response protocol based on JSON-RPC over WebSocket, as can be inferred by observing Figure 5. At the iMC interface, this protocol exposes the capability of negotiating media communications. The main entities and interactions associated to the iMC interface are the following:

Interacting with a Media Server: media negotiation and media exchange

In a media application involving a Media Server we identify three types of interacting entities:

- The Client Application, which involve the native media capabilities of the client platform plus the specific client-side application logic. The End-user in in contact with the Client Application.
- The Application Server, which is a container of procedures implementing the server-side application logic. The Application Server mediates between the Client Application and the Media Server in order to guarantee that the application requirements (e.g. security, auditability, dependability, etc.) are satisfied.
- The Media Server: which provides the media capabilities accessed by the Client Application.

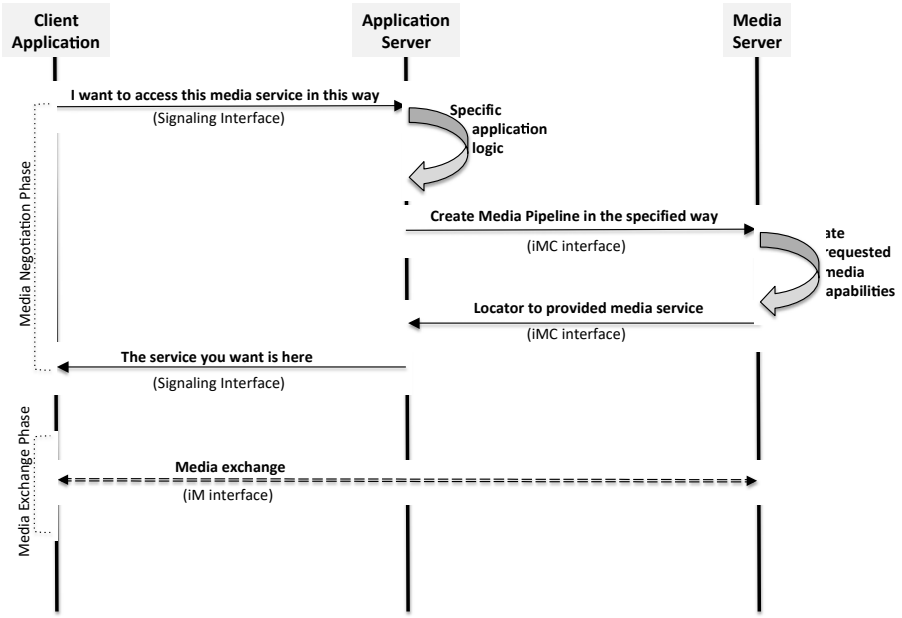

Figure 8. Typical interactions with Media Sever.
This diagram shows the typical entities and interactions involved in a media application based on a Media Server. In the case of NUBOMEDIA, the Application Server corresponds with the NUBOMEDIA PaaS component and, more specifically, with the container where the specific application logic is executing.

A shown in Figure 8, the interactions among these three entities typically occur in two phases: the media negotiation and the media exchange phase.

The media negotiation phase requires a signaling protocol that is used by the Client Application to issue messages to the Application Server requesting to access a specific media service. Typically those requests specify the specific media formats and transport mechanisms that the Client Application supports.

The Application Server provides semantics to these signaling messages by executing server-side application logic. This logic can include produces for Authentication, Authorization and Accounting (AAA), Call Detail Record (CDR) generation, resource provisioning and reservation, etc.

When the Application Server considers that the media service can be provided, and following the instructions provided by developers when creating the application logic, the Application Server commands the Media Server to instantiate and configure the appropriate media capabilities. Following the discussions in sections above, in the case of NUBOMEDIA, this means that the Application Server creates a Media Pipeline holding and connecting the appropriate Media Element instances suitable for providing the required service.

To conclude, the Application Server receives from the Media Server the appropriate information suitable for identifying how and where the media service can be accessed. This information is forwarded to the Client Application as part of the signaling message answer.

Remark that during the above mentioned steps no media data is really exchanges. All interactions have the objective of negotiating what's, how's, where's and when's of the media exchange.

The media exchange phase starts when both the Client Application and the Media Server are aware on the specific conditions in which the media needs to be encoded and transported. At this point, and depending on the negotiated scheme (i.e simplex, full-duplex, etc.) the Client Application and the Media Server start sending media data to the other end, where it is in turn received. The Media Server processes the received media data following the specific Media Pipeline topology created during the negotiation phase and may generate, as a result, a processed media stream to be send back to Client Applications.

Interactions with WebRTC clients

A specific relevant case for NUBOMEDIA happens when dealing with WebRTC [WEBRTC] clients. In this case, the negotiation phase takes place through the exchange of Session Description Protocol [RFC4566] messages that are sent from the Client Application to the Application Server through the signaling protocol and which are forwarded then to the Media Server instance through the iMC interface.

For WebRTC to be supported, a specific Media Element needs to be created supporting the WebRTC protocol suite. Following the accepted WebRTC naming conventions [ALVESTRAND], let's imagine that we call *WebRtcEndpoint* to such Media Element. In that case, the *WebRtcEndpoint* needs to provide SDP negotiation capabilities through the iMC interface as well as any other of the WebRTC features required for the negotiation (e.g. [TRICKLE ICE], etc.)

As a result of this negotiation, an SDP answer shall be generated from the MS *WebRtcEndpoint* to the Application Server. This will be forwarded them back to the Client Application that, in turn, can use it for starting the media exchange phase. This procedure is summarized in Figure 9.

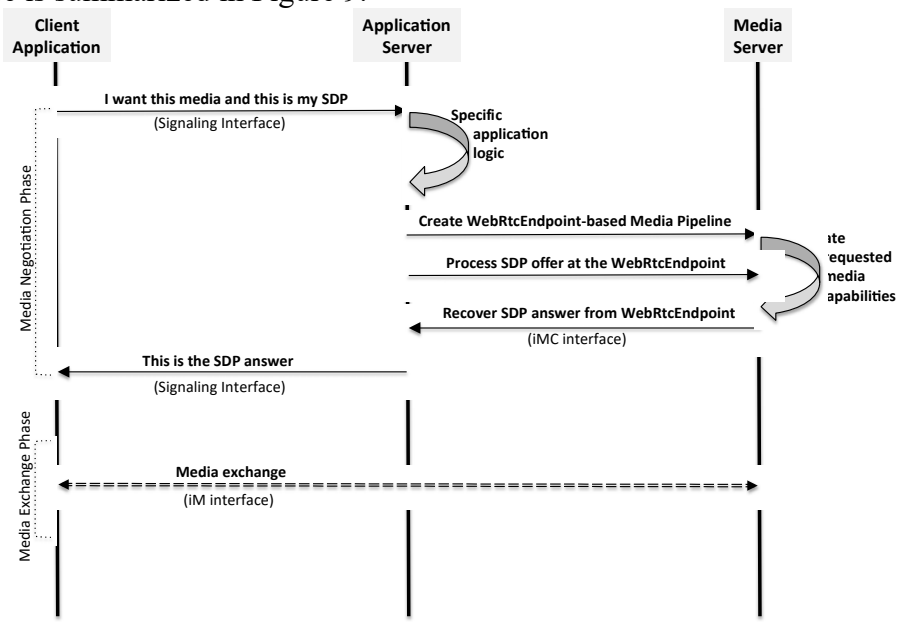

Figure 9. Interactions for establishing a WebRTC session

This diagram schematizes the different interactions taking place for establishing a WebRTC session between the Client Application and the Media Server.

4.5 Media Server Implementation

This section is devoted to introduce and explain how the Media Server has been implemented, so that it complies with the specified functional architecture depicted in Figure 1. The Media Server implementation details have been deeply influenced by a number of design decisions. For completeness, in the following section we introduce

such decisions so that their impact on the overall strategy and software architecture can be understood.

4.5.1 Enabling technologies for the NUBOMEDIA Media Server.

4.5.1.1 Kurento Media Server

Developing the Media Server from scratch would have been impractical. Most of the low level technologies required by the Media Server (e.g. codecs, data structures for media representation, thread management mechanism, etc.) already exist and re-implementing them would require a very relevant effort that would not provide any kind of novelty. Due to this, since the very beginning, the idea was to leverage some king of pre-existing technological framework providing such low-level technologies and, at the same time, giving the appropriate flexibility for creating the complex media orchestration flows and capabilities that the Media Server requires. The selected framework for this was the Kurento Media Server (http://www.kurento.org), also known as KMS.

KMS is a media server developed in the context of several research projects which include FIWARE (http://www.fiware.org) and AFICUS (http://www.ladyr.es/aficus). Kurento was conceived as a modular media server basing on pluggable extensions that are loaded upon server startup. In its original conception, KMS was devoted to providing flexible media processing capabilities basing on the notion of Filters (i.e. media processing modules). However, initial the real-time features of KMS were very limited and its stability and efficiency where not appropriate real-world applications.

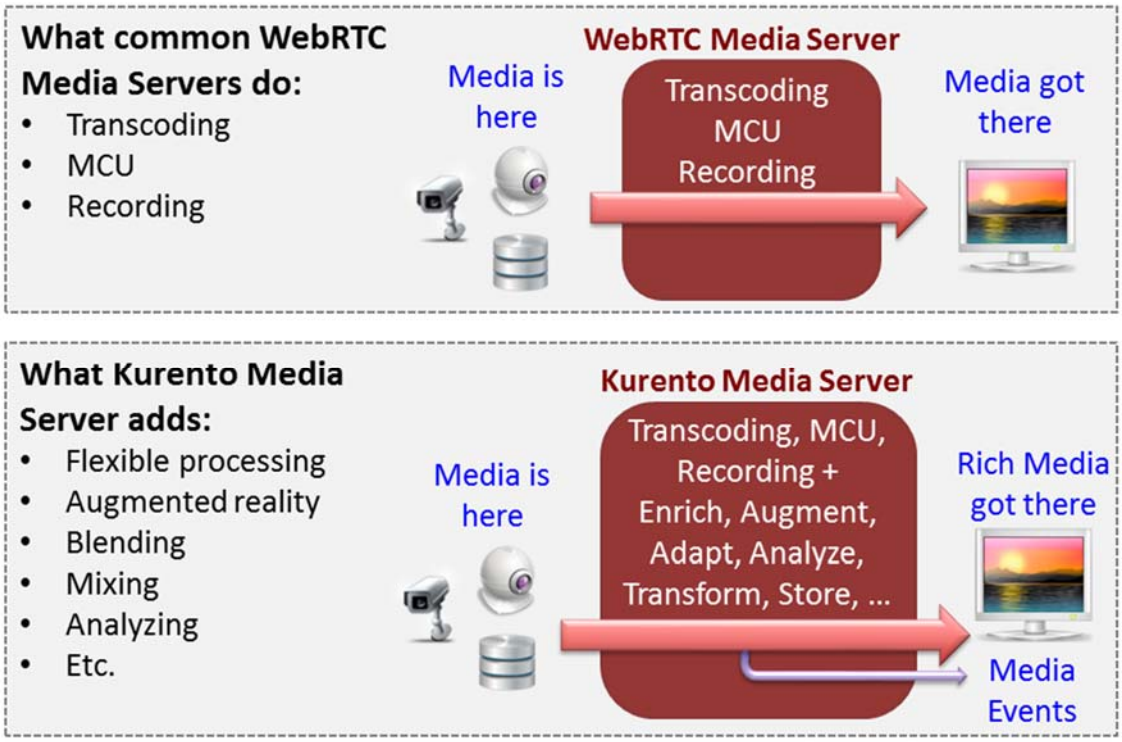

Figure 10. Capabilities of common media servers (top) vs. Kurento Media Server (bottom)

At it can be observed, Kurento Media Server incorporates advanced features for media processing including computer vision, augmented reality or any other media processing mechanism suitable to be implemented as a media filter.

For creating our Media Server, KMS has been revamped through a complete re-factorization and through the addition of advanced real-time media capabilities which include WebRTC, media instrumentation and monitoring, group communication capabilities, cloud adaption, etc. As a result, latest versions of KMS are fully adapted to

the NUBOMEDIA needs and, to all effects; KMS is now the NUBOMEDIA Media Server, reason why in this document we shall use Kurento, KMS and the NUBOMEDIA Media Server as synonyms

4.5.1.2 GStreamer

GStreamer [GSTREAMER] is an open source software project proving a pipeline-based multimedia framework written in the C programming language with the type system based on GObject. 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 allows a programmer to create media processing applications basing on the interconnection of media capabilities called *media elements*. Currently, in the GStreamer ecosystem there are hundreds of media elements providing disparate media capabilities such as codecs (i.e. encoders and decoders) for audio and video, players for media reading, recorders for media archiving, queues for media buffering, tees for media cloning, etc. Each GStreamer media 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 Figure 11. 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. 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.

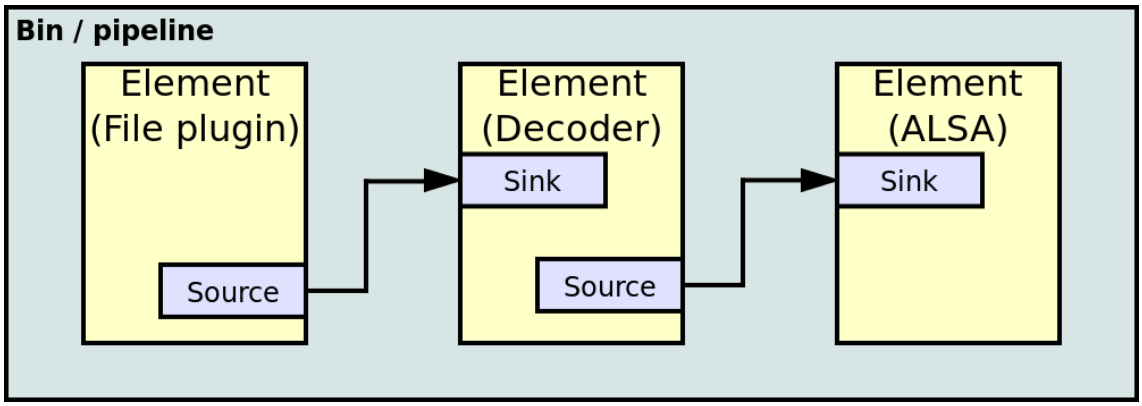

Figure 11 GStreamer pipeline as a chain of connected media elements
Source pads are connected to sink pads indicating how media streams flow through the processing graph. This pipeline exemplifies an audio player where an encoded file is first decoded and later feed to the sound driver.

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 1.8.3, 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)

Although GStreamer provides a bunch of interesting features, GStreamer media elements are very low-level and expose their media capabilities without relevant abstractions for developers. This means that, in GStreamer, developers cannot generate their media pipelines just by connecting freely the media elements they wish. These connections need to follow a careful design (you cannot connect any media element with any other) and cannot happen at arbitrary times (it does not manage automatically aspects such as key frames). Hence, for using GStreamer, developers need to have very deep knowledge on the internal workings of the media capabilities and be aware of very low level details including codecs, formats, threads, negotiations, queues, buffers, etc. Hence, the creation of a robust and useful pipeline in GStreamer may take several months of effort for a developer. In addition, GStreamer only provides media capabilities, but no mechanism for controlling them, which makes very difficult to integrate GStreamer real-world applications. As a result, developers having the ability of creating application basing on GStreamer are scarce and most of them work only with static pipelines (i.e. graphs of media elements that do not change on time) being

dynamic pipelines only used by GStreamer gurus due to their extremely high complexity.

In spite of these problems, GStreamer is a powerful framework for media processing and, due to this, KMS was writing leveraging GStreamer media capabilities. As a result, and given the discussions above, the NUBOMEDIA Media Server uses GStreamer as the media processing framework. In the next sections, we shall have the opportunity of understanding in detail the role of GStreamer in the NUBOMEDIA Media Server

4.5.1.3 Evolutions and extensions for the NUBOMEDIA Media Server

As introduced above, the implementation strategy for the NUBOMEDIA Media Server has been based on two axes:

- 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 provided, off the shelf, the features required by the NUBOMEDIA Media Server. For this reason, the implementation of this strategy has required the execution of a number of developments, adaptations and extensions, which are summarized below

4.5.1.3.1 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 some 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. At the time of this writing, around 75 contributions to the GStreamer project has been provided in the form of bug-reports, bug-fixes and featured enhancements.
- Lack of WebRTC 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.
- Lack of group communication capabilities. GStreamer group communication capabilities did not exist before NUBOMEDIA. The only pre-existing feature related to group communications were the video Composite and the audio Mixer, which could be considered as building blocks for mixing MCUs, but which required many bug-fixes and several adaptations. In addition, no pre-existing media routing capabilities where available. NUBOMEDIA has

introduced the ability onto GStreamer for providing SFU and MSM RTP topologies.

- Abstract media elements. In GStreamer media elements are not abstract and they cannot be connected freely to create arbitrary graphs (e.g. GStreamer sources can only be connected to one sink, GStreamer does not take care of format adaptation, etc.) For this, NUBOMEDIA has created a high-level notion of Media Element those fully abstract media complexities enabling developers to create dynamic arbitrary media processing graphs.

4.5.1.3.2 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 fully 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.

- Lack of group communication capabilities: KMS originally did not provide any kind of group communication capabilities. Now they are available through different types of topologies including SFU, MSM and Media Mixer.

4.5.2 Media Server Software Architecture

The functional system architecture of the Media Server has been introduced in Figure 1. For facing the objective of understanding the Media Server software architecture we start referring to that functional architecture. As stated before, the main functional components of the NUBOMEDIA architecture are two: The Media Server Manager (left side of the figure) and the media capabilities (right). These functional components are also mapped to the top-level elements of the software architecture. For understanding this, observe Figure 12. As it can be seen, the Media Server software artifacts are split into two families:

- Media Server Control Logic software artifacts. These correspond with the implementation of the Media Server Manager functional component. These artifacts have been developed in the C++ language from scratch consuming low level libraries such as boost [BOOST] or specialized libraries such as the *jsoncpp* parser [JSONCPP] but they do not leverage any GStreamer code.
- Media Capabilities software artifacts. These hold the media capabilities themselves. These artifacts make intensive use of GStreamer APIs and also include additional media management logic beyond the basic GStreamer mechanisms. All these artifacts have been written in the C programming language using the type system provided by GObject [GOBJECT]

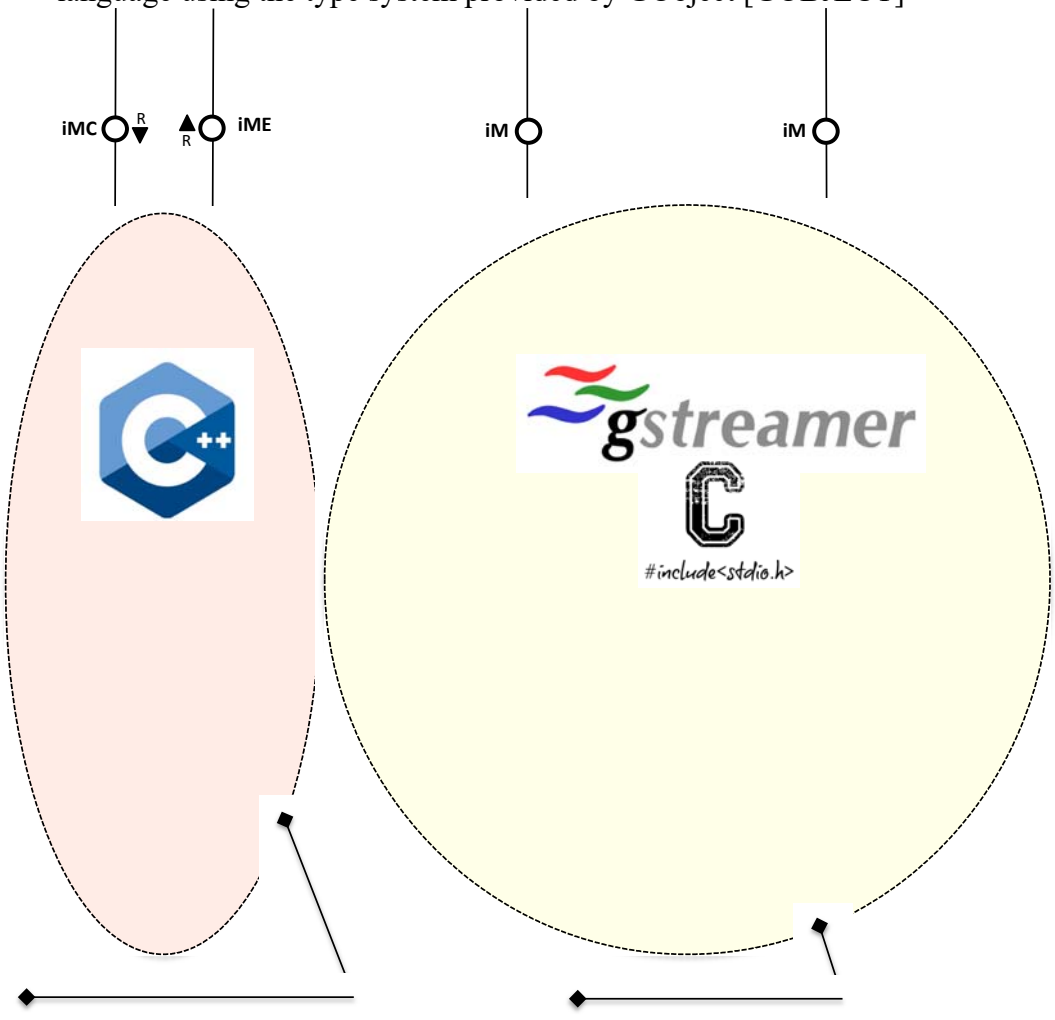

Figure 12. The software architecture of the Media Server is split into two main components.

The Media Server Control Logic Software Artifacts (left) comprise all software artifacts devoted to the orchestration of the media capabilities and, in particular, the artifacts implementing the Media Server Manager function. The Media Capabilities themselves consist on a number of software artifacts providing the Media Pipelines and Media Elements having the media transport, processing and manipulation features. These later are strongly based on GStreamer APIs and extensions.

4.5.3 Media Server Control Logic Software Architecture

The Media Server Control Logic Software Architecture has been designed as an implementation of the Media Server Manager functional module. The Media Server Manager functional architecture is depicted in Figure 5. In relation to its software architecture, the Media Server Manager has been fully developed in C++. The code structure for the corresponding classes is introduced in the following sections.

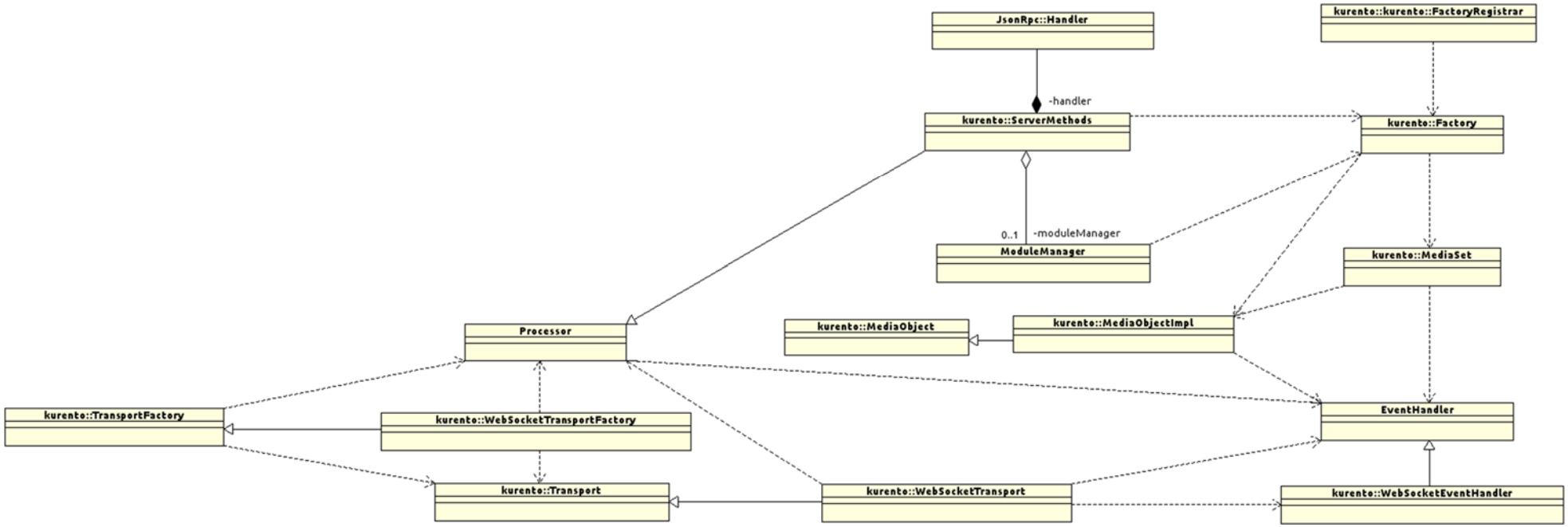

Figure 13. Software architecture of the Media Server Module Manager

This UML class diagram shows the different classes comprising the Media Server control mechanisms.

4.5.3.1 WebSockets Manager

This component has been implemented through three classes:

- WebSocketTransportFactory
- WebSocketTransport
- WebSocketEventHandler

This component has been developed based on the websocketpp library available in https://github.com/zaphoyd/websocketpp.

The WebSocketTransportFactory creates a WebSocketTransport associated to a KMS Processor. In that way, when a request is received through the websocket, the WebSocketTransport uses the Processor to respond it.

The WebSocketEventHandler class is used to successfully redirect events to correct websocket. When a subscription request is received, a WebSocketEventHandler is created associated to sessionId and the event. When an event is raised it forwards the event to the proper connection associated with the sessionId.

4.5.3.2 JSON-RPC Manager

ServerMethods class contains a JsonRpc::Handler this class allows registering methods that will be called using json. When a request is received, it is passed to the handler. Handlers parse the requests handling errors in json-rpc protocol and then dispatch the request to the registered method. When the method is executed, the handler generates the proper response in jsonrpc (response or error if a JsonRpc::CallException is raised during the process).

4.5.3.3 Server Methods Dispatcher

The ModuleManager component is responsible for find all the Kurento Modules available in the configured directories.

This component loads the module in memory in order to be available to use it runtime and it registers the different factories existing in each module.

4.5.3.4 Module Manager

ServerMethods class contains a JsonRpc::Handler this class allows registering methods that will be called using json. When a request is received, it is passed to the handler. Handlers parse the requests handling errors in json-rpc protocol and then dispatch the request to the registered method. When the method is executed, the handler generates the proper response in jsonrpc (response or error if a JsonRpc::CallException is raised during the process).

4.5.3.5 Factory Manager

This component has been implemented through two classes:

- FactoryRegistrar.
- Factory.

The Factory class is able to create any MediaObject available in the KMS from the factory name. Also it is responsible to return an existing MediaObject from its objectId.

The FactoryRegistrar class is used to keep a list of Factories for all the available factories in KMS.

Each module can register Factory child classes to FactoryRegistar, this is how kurento capabilities can be extended.

4.5.3.6 Media Set

This component is responsible for keep a list of created MediaObjects related with its session. For each object, the MediaSet stores the number of session that is using this MediaObject and it is able to delete an unused MediaObjects.

This component is also responsible to receive the keepsalive from the clients to avoid sessions to be destroyed and to execute the garbage collector in the configured time

4.5.3.7 Server Manager

The ServerManager is a MediaObject that contains info about the running instance of KMS. You can get several data from this component:

- The kmd file related to a module giving its name.
- The pipelines running into KMS.
- The available sessions.
- Metadata from KMS.
- Info related to KMS like version, available modules, ...

4.5.3.8 Media Object Manager

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.

4.5.3.9 Media Object

MediaObject is the base class for all the objects remotely managed. All this objects livecycle is managed by MediaSet.

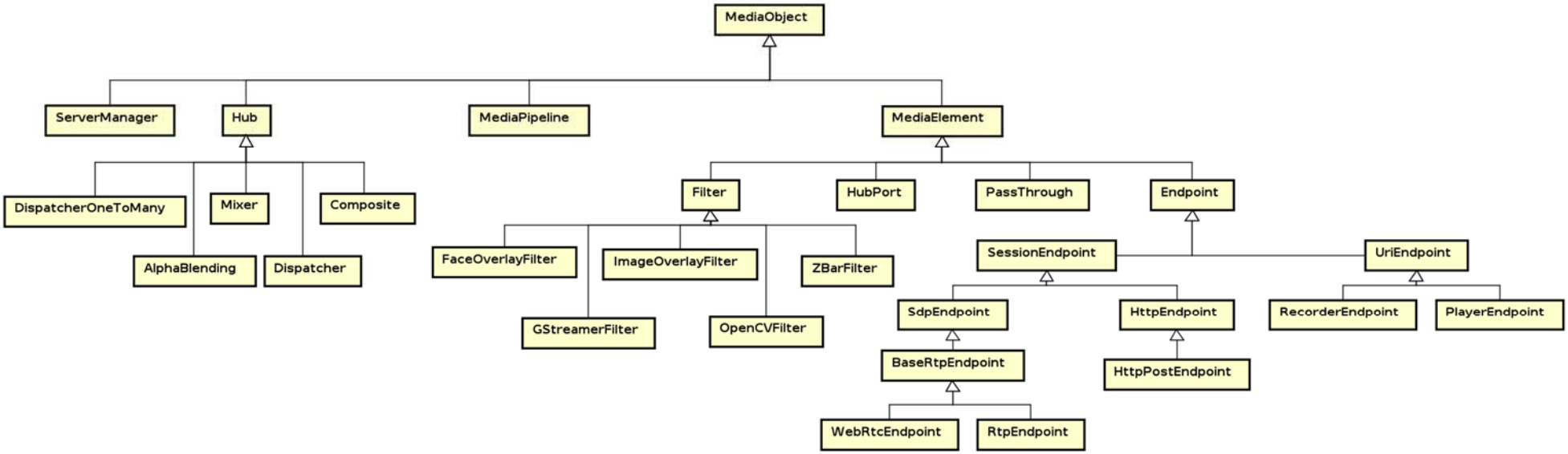

Figure 14. MediaObject Inheritance

4.5.4 Media Objects and media capabilities

In this section we try to introduce the Media Object hierarchy that the Media Server uses for controlling media capabilities and its relation with the media capabilities themselves. For achieving this objective, it is important to introduce a naming convention that shall be helpful for reasoning about the different involved concepts.

In sections above, we have referred to the media capabilities of the media server as "Media Elements" and "Media Pipelines". These concepts are implemented inside the Media Server through two types of software entities: Media Objects, implemented in C++, and GStreamer bins, implemented in C/GObject. In both cases, a notion of media element and media pipeline also exists. Hence, we have three different entities that we can call media element and media pipeline, but the three refer to completely different constructs and have different properties and functions. For clarifying, let's establish the following conventions:

- Media Element and Media Pipeline (with first letter in upper case) are Media Server functions (i.e. are functionalities of the Media Server exposed to developers) Hence, Media Element and Media Pipeline refer to abstract concepts that developers use for creating their applications
- Media Objects are C++ proxies to the real media capabilities. Among Media Objects, we can find the MediaElement and MediaPipeline classes. For avoiding ambiguity, these are called C++ media elements and C++ media pipelines.
- The real implementation of the media capabilities are called C media element and C media pipelines (as they are written in C). These are also called Media Element and Media Pipeline implementations along the text.

In addition, inside GStreamer there is an additional notion of media element and media pipeline. In both cases, they are named as GStreamer element and GStreamer pipeline.

All in all, the terms Media Element and Media Pipeline may refer to all the internal software objects that the Media Server requires for implementing the function of a specific (conceptual) Media Element or Media Pipeline.

Following this, as specified in sections above, the Media Server media capabilities consist on Media Elements that are interconnected through some kind of topology to form Media Pipelines. These capabilities are implemented in low level C and comprise complex combinations of GStreamer components. Exposing such complexity to developers would be impractical. Due to this, for every existing Media Element type the Media Server Manager holds a Media Object class that abstracts the Media Element complexities and enables exposing them through a coherent API compatible with iMC and iME request/response JSON-RPC models. In other words, Media Object types map the simple iMC/iME interfaces used by developers to the complex iMCI interfaces that are exposed by the Media Element implementations. Hence, for every JSON-RPC primitive exposed by the iMC interface in relation to a specific media capability, the corresponding Media Object shall expose an equivalent interface to the Dispatcher and shall provide semantics to these interfaces by executing whatever actions are necessary into the real capability through the iMCI interface.

The implications of this are straightforward: the Media Server API is constrained to the capabilities exposed by the different Media Object classes which are supported. In other words, application developers only interact with the media server through the iMC/iME interfaces following the API exposed by the available Media Objects. Hence,

application developers need to understand Media Object interfaces for being able to leverage all the features of the Media Server.

With this aim, we have created Figure 15 where the whole built-in Media Object hierarchy is depicted. In this context, built-in means that all these Media Objects are provided off-the-shelf by the media server. In addition to built-in capabilities, the Media Server can hold custom modules, which are introduced later in this document.

This Media Object hierarchy has a direct mapping with the corresponding media capabilities meaning that, for each specific Media Object instance, a media capability (e.g. Media Element, a Media Pipeline, etc.) implementation shall be associated and controlled by the Media Object. This also evidences those media capabilities implementations need to have an equivalent hierarchical structure, which is introduced later in this document.

Coming back to Figure 15, we observe that the MediaObject class, which is abstract, has four descendants, which are associated with the main 4 APIs exposed through the iMC interface:

- MediaPipeline: This class exposes the ability of managing Media Pipelines. It exposes primitives for managing pipeline members (i.e. factory methods, query lists of children, etc.) and pipeline lifecycle (e.g. errors, deletion, etc.) This class is specific (not abstract) meaning that it can be instantiated to represent a real Media Pipeline.
- MediaElement: This class exposes the ability of managing Media Elements. Its main primitive is “connect”, which enables Media Elements to be connected among each other. This class is abstract (it cannot represent a real media capability instance) but it serves as base class for the inheritance hierarchy of all specific Media Elements.
- Hub: This class holds basic Media Hub capabilities, which enable the management of group of streams. This class is abstract (it cannot represent a real media capability instance) but it serves as base class for the inheritance hierarchy of all specific Media Hubs.
- ServerManager: This is a special class used for instrumenting the Media Server. It makes possible to access Media Server runtime information (e.g. references to the existing pipelines) and also Media Server configuration data.

Following with the architectural descriptions presented above, the MediaElement class has two descendants that correspond with the two main types of Media Elements: Endpoint and Filter.

- Endpoints are Media Elements providing the ability of communicating media with the external world (out of or into the Media Pipeline)
- Filters are Media Elements providing media processing capabilities.

As we can observe, some specific realizations of Filter are the following:

- FaceOverlayFilter: This filter detects faces and overlay images on top.
- GstreamerFilter: This filter provides access to low level GStreamer capabilities. It is possible to use any filter available in gstreamer packages through this MediaElement in the same way that would be used to instantiate the filter using *gst-launch* *<http://rpm.pbone.net/index.php3/stat/45/idpl/19531544/numer/1/nazwa/gst-*

launch-1.0>. Then only restriction is that the elements need a src pad and a sink pad.

- ImageOverlayFilter: This filter makes possible to overlay images on top of videos.
- OpenCVFilter: It is an abstract class. This element is the base of the OpenCV Modules. It is used to call to process method of this kind of external modules for each new frame.
- ZbarFilter: This filter detects QR or bar codes and publishes, through an event, their values to the application.

In turn, the specific realizations of Endpoint are:

- RtpEndpoint: Endpoint that provides bidirectional content delivery capabilities with remote networked peers through RTP protocol.
- WebRtcEndpoint: This type of Endpoint offers media streaming using the WebRTC protocol.
- HttpPostEndpoint: This element provides access to an HTTP file upload function. This type of endpoint provides unidirectional communications.
- PlayerEndpoint: This element allows inject media to KMS from different video sources like files, urls or rtsp cameras in different formats.
- RecorderEndpoint: This element allows storing media from the KMS in a file or in a url.

To conclude, the specific built-in Hubs are the following:

- Composite: This element mixes all their input flows in an only flow. The input flows are shown in a grid
- Dispatcher: This element allows to route media between ports connected to it.
- DispatcherOneToMany: This element allows routing the media from one of the port connected to the rest of connected elements. The source port can be changed in runtime.
- AlphaBlending: This mixer element allows mixing different flows in only one choosing the position and the alpha level of each flow.

As explained above, every MediaObject class maps to a specific C implementation, which holds the real media capabilities, and which is controlled by the MediaObject. Table 1 shows this mapping for the main MediaObject types. The interested reader can refer to the description of the specific media capabilities provided in sections below for having a detailed description of the internal workings and exposed features of every Media Object.

Media Object type	Media Capability type
ServerManager	NA
MediaPipeline	KmsMediaPipeline
Endpoints	
RtpEndpoint	KmsRtpEndpoint
WebRtcEndpoint	KmsWebRtcEndpoint
HttpPostEndpoint	KmsHttpPostEndpoint
PlayerEndpoint	KmsPlayerEndpoint
RecorderEndpoint	KmsRecorderEndpoint
Filters	
FaceOlverlayFilter	KmsFilter with a

	KmsFaceOverlayFilter inside
GstreamerFilter	KmsFilter with any GStreamer filter inside
ImageOverlayFilter	KmsFilter with a KmsLogoOverlayFilter inside
OpenCVFilter	KmsFilter with a KmsOpenCVFilter inside
ZbarFilter	KmsFilter with a GstZbar filter inside
Hubs	
Composite	KmsCompositeMixer
Dispatcher	KmsDispatcher
DispatcherOneToMany	KmsDispatcherOneToMany
AlphaBlending	KmsAlphaBlending

Table 1. Mapping of Media Object types (written in C++) with every media capability type (written in C/GObject)

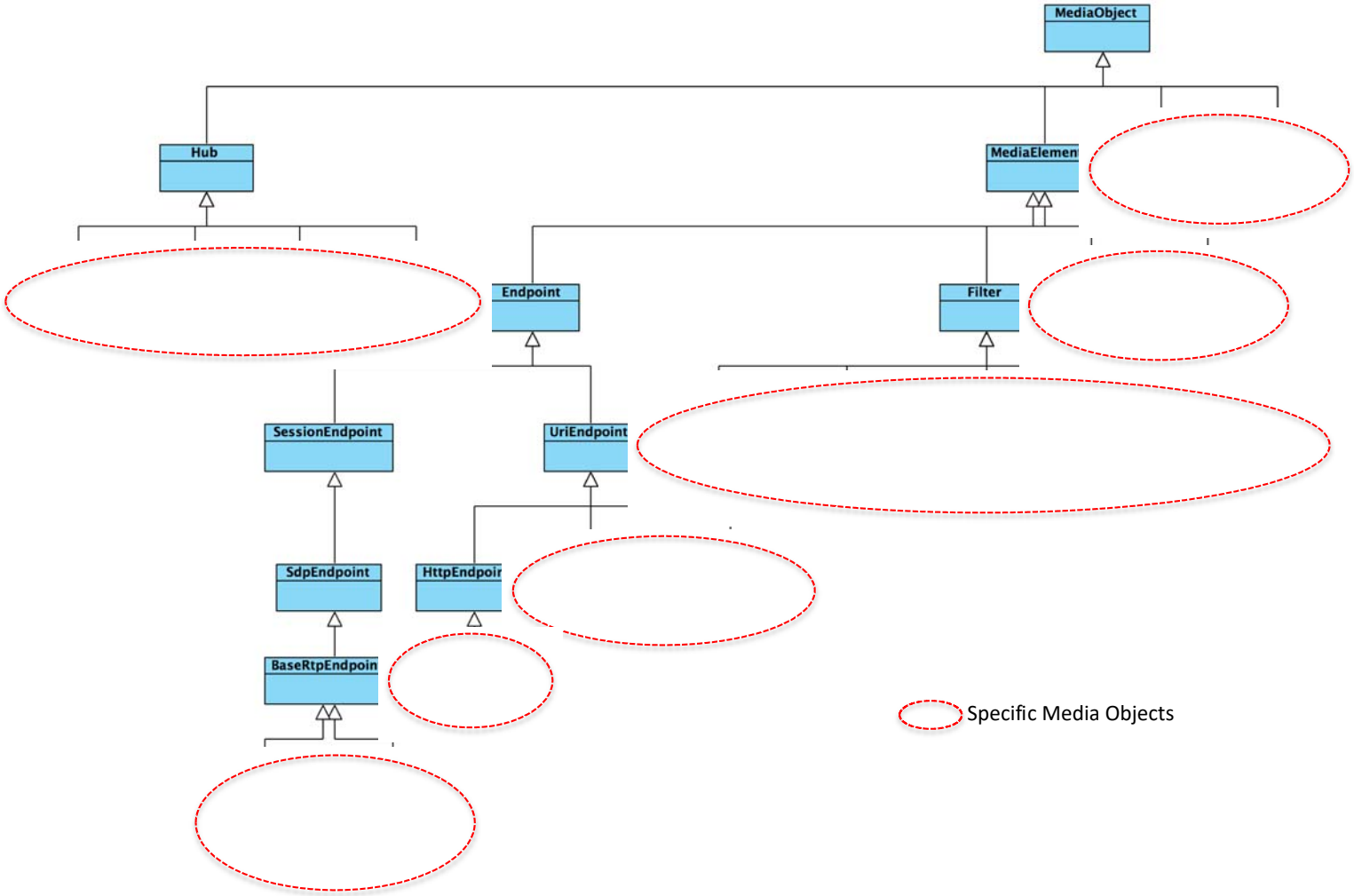

Figure 15. Inheritance hierarchy of the built-in C++ Media Objects
These objects are made available by the Media Server.

4.5.5 Software Architecture of the Media Capabiities

Once with have introduced the Media Object proxy classes, which provide the interfaces for the Media Element and Media Pipelines used by application developers, we need to understand how the real media capabilities are implemented. As introduced above, these have been developed in C/GObject leveraging GStreamer APIs and components following the scheme depicted in Table 1. The main classes among them and their details are introduced in the following sections.

4.5.5.1 KmsElement

This is the base class of all Media Element implementations. It extends the GStreamer class GstBin, which in turn extends the GStreamer class GstElement. This inheritance hierarchy is shown in Figure 16. GstElement represents a low-level GStreamer media element, which is a low-level specific and non-abstract capability. GstBin is a GStreamer object representing a chain of GstElements, which is typically called a *bin* in GStreamer jargon. This chain may be dynamically modified so that the internal function it performs is adapted, in an abstract way, to the requirements of the application using it.

On top of GStreamer GstBin, KmsElement provides two key features: the ability of managing *pads* and the ability of managing *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. In addition, as a bin, the KmsElements can contain chains of GStreamer elements, which are the ones in charge of the real media processing capabilities. This is the case of the Agnostic bin, a specific bin that exists inside each KmsElement, which in charge of managing the media transformations and adaptations needed for enabling seamless interconnectivity of KmsElements.

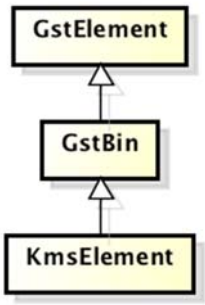

Figure 16. KmsElement UML class diagram

KmsElement inherits from the GStreamer GstBin object, which represents a chain of GStreamer elements, which in turn inherits from GstElement, which represents a GStreamer element itself.

The runtime structure of a KmsElement, with the internal GStreamer elements that it contains, is shown in Figure 17. This type of figure is based on GST DOT format and is generated automatically by GStreamer under request. GST DOT figures are color-coded for clarity. The codes can be summarized as follows:

- 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 or bins.
- Grey “cloud” represents the part of the KMS element where the media processing takes place.
-

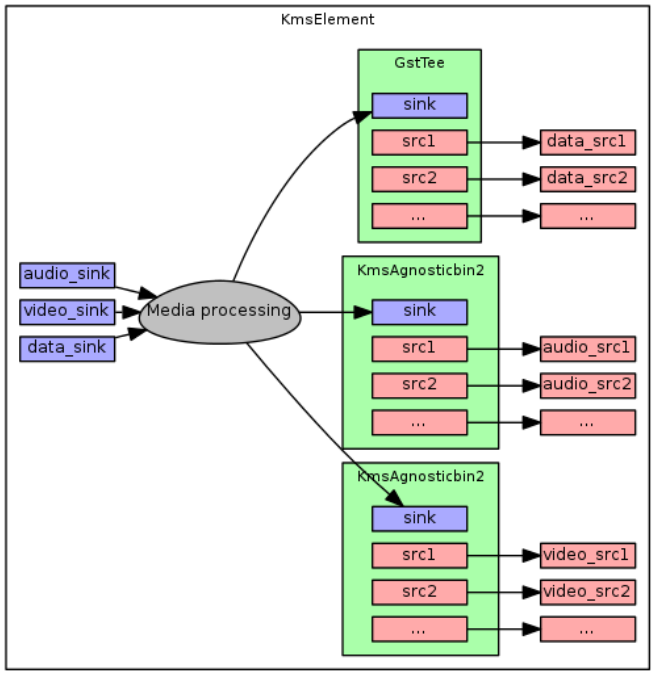

Figure 17. KmsElement runtime structure
This chart shows the internal GStreamer objects comprising it in GST DOT format In this format, solid shapes represent GStreamer objects and arrows represent media flows. Code colors are also useful. Violet boxes represent GStreamer sink pads that capable of receiving media. Red boxes represent source pads, that have the ability of sending media. Green boxes represent GStreamer elements or bins. Gray clouds represent GStreamer elements where the specific processing logic of the KmsElement is implemented.

Hence, we can observe that, after receiving audio, video and/or data through its sink pads, a KmsElement performs some type of media processing. When that processing is over, video and audio are sent separately to a special GStreamer element, the KmsAgnosticBin2 (also called KmsAgnosticBin or just AgnosticBin for brevity). The AgnosticBin is a tailor-made element which is in charge of adapting between different media formats and performing the codec adaptation 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. AgnosticBin is one of the main innovations of the Kurento Media Server in relation to GStreamer architecture given that it enables developers to interconnect KmsElements with full freedom as it clones and adapts the media flows to the specific requirement of each of the connected consumers. Its details are introduced later in this document. Data is just passed through a tee element that allows send a stream to multiple elements.

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 codec 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 has one sink pad for audio, one sink pad for video and one sink pad for data, implying that only one stream of each type can be managed by on element. In our architecture there is a different kind of elements, called *Hubs*, which can manage multiple streams. They are introduced later in this document. On the other hand, there is no limit on the number of source pads a KmsElement may have. Hence, KmsElements are able able to

serve to as many sink elements as needed, being the only limitation the ones associated to the available computing resources on the executing hardware.

KmsElement is an abstract class, meaning that it cannot be directly instantiated. The real Media Element implementations in our architecture are specific descendants, each of which need to implement their own media processing logic (i.e. the grey "cloud"). Media Element implementations need also to signal when they are ready to receive audio and video, so the sink pads can be created, and to indicate the its parent which type of GStreamer media formats the specific KmsElement accepts. As an example, if a specific Media Element class expects to receive video in raw format (i.e. not encoded), it has to indicate so.

4.5.5.1.1 Data Streams and MetaData

We need to specially mention data streams that it is an important addition done by the Nubomedia project to KmsElement. This addition allows MediaElements to process data streams, adding a proper sink pad and also a tee at the output to allow sending the stream to multiple elements. Processing of data streams is done by subclasses, they can: create new data (eg: doing some detection on a image and notify downstream elements) process data received from other elements (eg: a face detection has been done by an upstream element and this data is used to draw a mask), forward the data, modify data and re-send it (eg: get a stream where some detection has been done and complete the information provided with more information).

Media streams are used in the same way as other media stream. A MediaElement that wishes to make use of data streams only need to implement a GStreamer element that can be connected to MediaElement data pads.

Working with them from the GStreamer perspective is just the same as working with audio or video. Data should be wrapped in a GstBuffer structure before sending it. Then, calling the function gst_pad_push using the appropriate data pad will send the data. Other option is to create a bin that includes a appsrc element. This element accepts GstBuffer and send them to the peer elements. An example of how to implement a media element that can send data can be seen here (https://github.com/nubomedia/kms-datachannelexample/blob/master/src/gst-plugins/kmssenddata.c#L127). Data reception is quite similar, when a peer element pushes GstBuffer with data, this data arrives to the next element sink pad. Then, the data can be get from the buffer and interpreted as expected (eg: As a string of characters). Here (https://github.com/nubomedia/kms-datachannelexample/blob/master/src/gst-plugins/kmsshowdata.c#L71) there is an example of how to get data from a pad.

Even data streams could be useful, there can be problems while synchronizing data streams with audio/video streams. For this purpose, MetaData can be added to audio and video buffers, this way synchronization between data stream and media stream is done automatically as the data "travels" alongside with the media. The only limitation that the MetaData has is that its usage is restricted inside the MediaServer because Endpoints cannot send it. To solve this problem a filter that extracts MetaData and re-sends it using the data stream can be used (same on the opposite direction, getting a data stream an adding it as MetaData). Combinations of data streams and MetaData included in media buffers should cover a wider spectrum of applications that require data

streams. For example, face detection data will be probably included as MetaData instead of as a separate data stream, this way the synchronization is perfect.

Working with metadata requires a GStreamer element that can work with the media that carries the metadata. For inserting data in the stream buffers, you need access to the GstBuffer structure (something common when working with GStreamer elements). Kurento has functions that allows inserting GstStructure data, that is a mutable map indexed by strings. For inserting data in the buffer the function kms_buffer_add_serializable_meta should be called, with the buffer to include the information into and the GstStructure with the data. This function takes care of data already set to the buffer and mixes it properly: updating values that were already set and adding new values. There is an example of a plugin that inserts metadata into buffers here (https://github.com/nubomedia/kms-datachannelexample/blob/master/src/gst-plugins/kmsfacedetectormetadata.c#L221).

To get the data that has been associated with a buffer, function kms_serializable_meta_get_metadata has to be called. This returns the GstStructure that is currently associated with the buffer as it was set by previous calls to kms_buffer_add_serialiable_meta. An example plugins that makes use of metadata in the buffers can be found here (https://github.com/nubomedia/kms-datachannelexample/blob/master/src/gst-plugins/kmsimageoverlaymetadata.c#L214).

4.5.5.2 KmsAgnosticBin

This GStreamer bin, as explained in the previous section, is responsible for the seamless interconnection of all media elements. It is thanks to this GStreamer bin, 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 RTP - H264 encoded - endpoint, without requiring any special actions from the programmer.

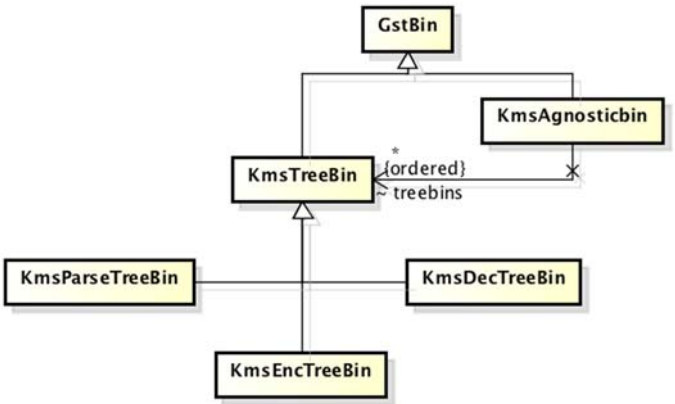

Figure 18: KmsAgnosticBin UML diagram

Every KmsElement has two KmsAgnosticBins, one for audio and one for video. As it can be observed in Figure 18, the KmsAgnosticBin is a GStreamer bin that comprises four different types of capabilities, which are themselves bins:

- The KmsTreeBin enables to create tree graphs of elements. In other words, makes possible to the KmsAgnosticBin to have several sources of media.
- The KmsParseTreeBin is a bin having the ability of analyzing media stream. This bin "understands" media formats by inspecting the stream bits and completes the caps (media description) that downstream elements has notified.

- The KmsEncTreeBin has the ability of encoding media streams to appropriate target formats.
- The KmsDecTreeBin, in turn, provides the capability of decoding media streams from a wide spectrum of source formats.

Thanks to these capabilities, the KmsAgnosticBin allows connecting elements independently on their media capabilities (i.e. codec, frame-rate, resolution, etc.). The set of all these capabilities (aka formats) is called *caps* in the GStreamer framework. The KmsAgnosticBin is capable of distributing media with different caps and to several sinks. For this purpose, caps from the incoming media (the one offered at the KmsElement sink) are compared against the ones required by the connected elements (the one given through each the KmsElement source to their corresponding sinks). If incoming media at the sink has different caps than the outgoing media at the sources, the appropriate transcodings and adaptations are performed. Hence, media may be transformed in order to comply with the caps requirements from source pads. Provided an element may have more than one source pad, and taking into account that coding is a costly process, the agnostic bin optimizes codec operations. Intuitively this means that the KmsAgnosticBin only performs once a given de-coding or en-coding, reusing it whenever it is necessary. The internal structure of the agnostic bin is depicted in Figure 19. We have split it into 7 blocks in order to make possible an in-depth explanation of its internal workings:

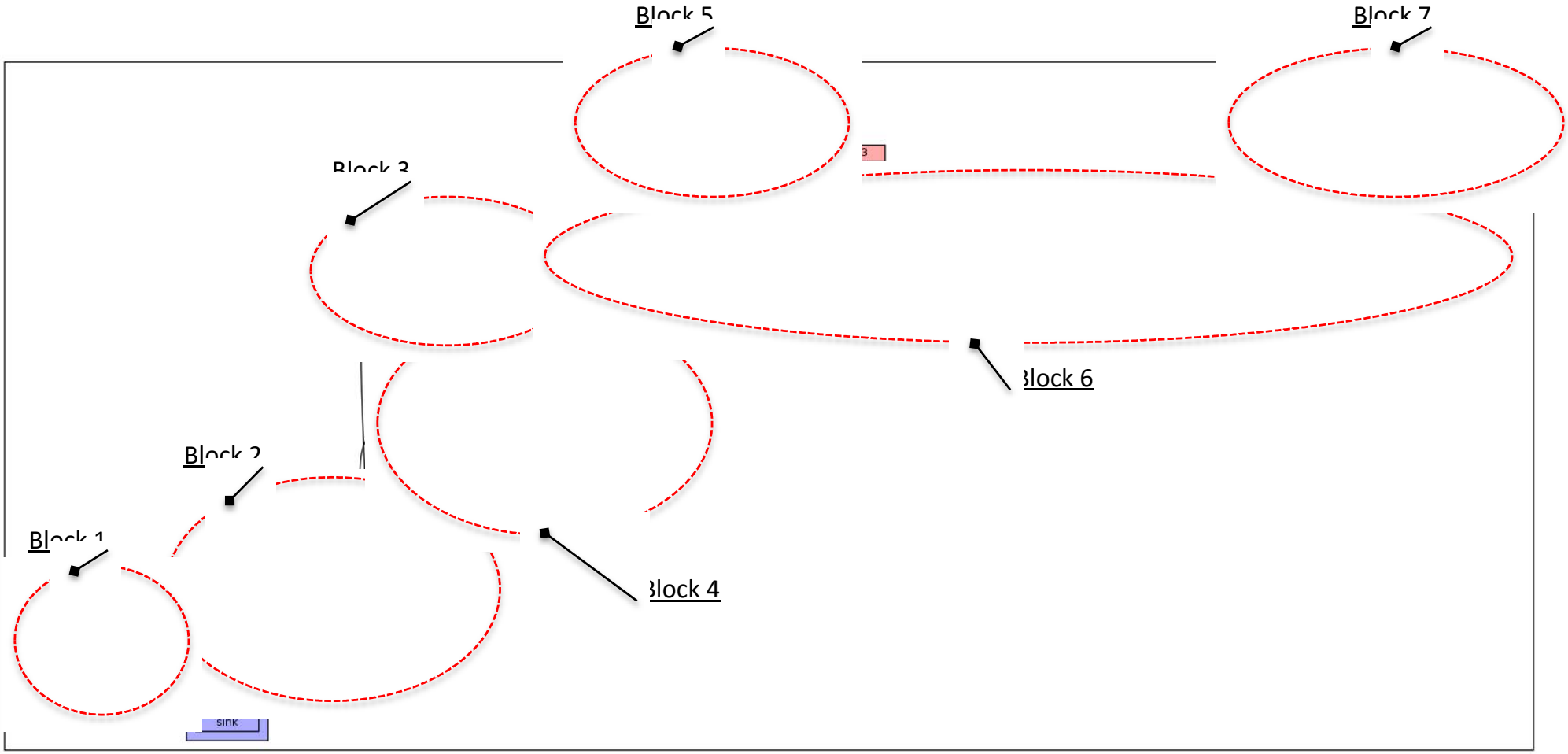

Figure 19: KmsAgnosticBin internal structure
Every KmsElement has two KmsAgnosticBins: one for audio and one for video.

- In Block 1 media is received from the KmsElement processing logic, as shown on Figure 17. This media is cloned through a GstTee (the GStreamer media element enabling to clone a sink input into many sources outputs). One clone goes to a dummy sink, called GstFakeSink, which is required by GStreamer internal logic when the KmsElement is not connected to any target sink.
- Block 2 shows the KmsParseTreeBin component, which introduces a PARSER GStreamer element for analyzing incoming media stream caps and, hence, complete the description provided.
- Block 3 comprises a source which is cloning the original media stream received from the sink. That source shall acts as a pass-through for target KmsElements wishing to receive the media in the original format. KmsAgnosticBin may create as many instances as Block 3 as required when many target KmsElements request the pass-through media. Remark that Block 3 has a GstQueue element, which is the GStreamer element in charge of queuing media and pushing it to a source on an independent thread. This mechanism guarantees that target KmsElements receiving the media cannot interfere with the source KmsElement by blocking the media stream thread.
- Block 4 accounts for the KmsDecTreeBin. When the KmsAgnosticBin detects codec incompatibilities, it performs a transcoding being the decode operation the first step. Due to this, this bin has a GStreamer DECODER element.
- Block 5 is in charge of providing raw media to target connected KmsElements. For this, these KmsElements must require raw media in their caps. Again, as with Block 3, this is performed through a GstTee and a GstQueue.
- Block 6 performs the encoding operations of a transcoding. Whenever a target KmsElement requires media in an incompatible codec, a KmsEncTreeBin shall be added to the KmsAgnosticBin performing the media encoding from the raw media. This encoding may also require rate or scale adaptations that are implemented by the corresponding GStreamer elements.
- Block 7, as block 5 and 3, represents the capability of providing an encoded media stream to as many sinks as necessary as they can be repeated multiple times.

All the elements present in Figure 19 are managed by an algorithm that inserts only the required parts at the appropriate moment.

- Initially, only Block1 is present, until something is connected on sink pad and peer capabilities are notified (this is the normal GStreamer procedure). The capabilities announced by the peer element are analyzed and then a proper parser is chosen. Creating and adding Block2.
- Once Block2 generates an improved capabilities description, agnosticbin is ready to be connected to downstream elements. In case downstream elements were connected previously to this point, they are connected at this moment following the same mechanism (described bellow).
- When a new sink pad is requested and connected, this means that a new media element is connected downstream and media should be fed to it. First, we need is to know what media does this element need. For this, a query is sent downstream through this pad to know what caps are accepted. Then, using this information we try to connect (and generate if required) to the appropriate internal block. We can find following situations:

- Requested caps can are the same as the input caps. In this situation we create a Block3 to perform the connection. The flow diagram illustrating how the agnosticbin behaves in this case is depicted in Figure 20
- If the requested caps are raw and the input is not raw, we need to decode the stream, creating, if it is not already created, the Block4 and then a Block5 where the output is connected. Figure 21 illustrates how this process takes place.
- Finally, if the media is encoded and it does not match the input, we need to create a Block4 (to get decoded media) if it is not already created, and then create a proper Block6 if there is none that is already encoding to the proper format (note the there can be multiple Block6 if multiple encoding formats are required). Then we need a Block7 to connect the output properly. This is illustrated on Figure 22

As a result, the KmsAgnosticBin is conceptually a tree of GStreamer media elements which is capable of providing pass-through media, raw media or transcoded media to an arbitrary number of sinks (i.e. target KmsElements).

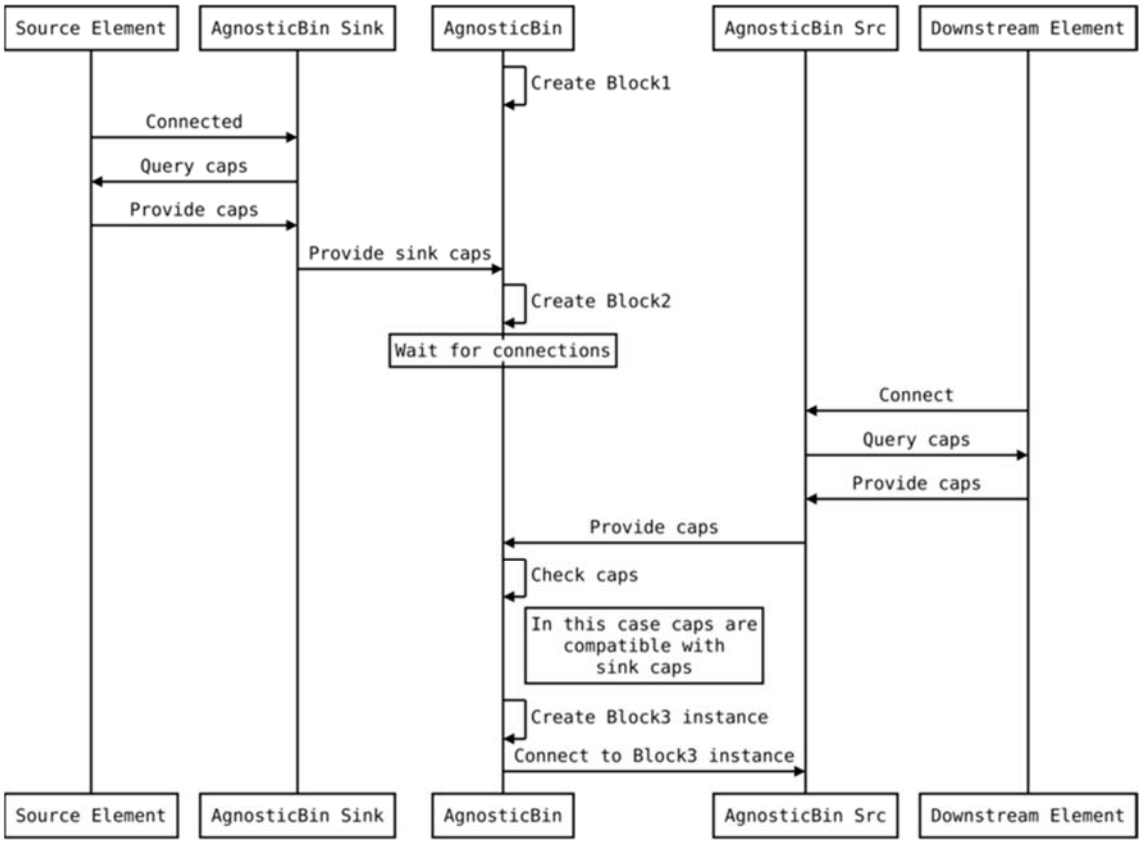

Figure 20. This diagram shows the process executed by agnostic bin to perform its internal connections and blocks construction.

In this case the capabilities (i.e. caps) requested by the downstream element are compatible with the input caps. First, the element that feeds media to agnosticbin (Source Element) needs to be connected and provide the capabilities of the media. This will create Blocks 1 and 2. Then agnosticbin waits for downstream elements to feed media to. When a new element is connected, their caps are queried in order to know the format that that agnostic should be feed to it, in this case the requested media is fully compatible with the one provided, no transcoding is required. Then, a Block3 is created and connected to provide media to downstream element. In this specific example, the caps presented by the Source Element are represented by the following string "video/x-vp8, profile=0, streamheader=< 4f5650383001010003... > ,width=(int)854,height=(int)480,pixel-aspect-ratio=1/1,framerate=23/1". The caps requested by the Downstream Element are "video/x-vp8" and the resulting final caps provided by the AgnosticBin are "video/x-vp8, profile=0, streamheader=< 4f5650383001010003...>,width=(int)854,height=(int)480,pixel-aspect-ratio=1/1,framerate=23/1"

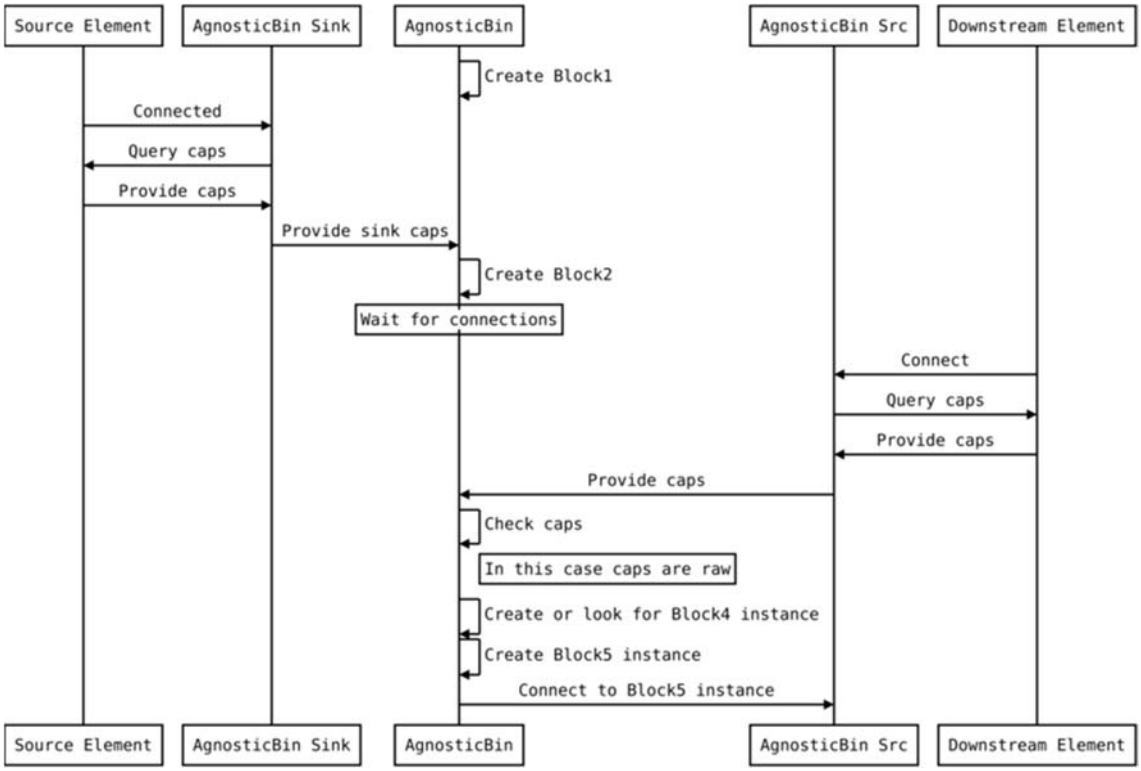

Figure 21. This diagram shows the process executed by agnostic bin to perform its internal connections and blocks construction.

In this case the capabilities requested by downstream element are raw and a decoding of the input media should be done. First, the element that feeds media to agnosticbin (Source Element) needs to be connected and provide the capabilities of the media. This will create Blocks 1 and 2. Then agnosticbin waits for downstream elements to feed media to. When a new element is connected, their caps are queried in order to know the format that that agnostic should be feed to it, in this case the requested media is raw. If previously Block4 has been created, it is used directly, otherwise it is created. Then, a Block5 is created and connected to provide media to downstream element. In this specific example, the caps presented by the Source Element are represented by the following string "video/x-vp8, profile=0, streamheader=< 4f5650383001010003... > ,width=(int)854,height=(int)480,pixel-aspect-ratio=1/1,framerate=23/1". The caps requested by the Downstream Element are "video/x-raw" and the resulting final caps provided by the AgnosticBin are "video/x-raw, format=(string)I420, width=(int)854, height=(int)480, interlace-mode=(string)progressive, pixel-aspect-ratio=(fraction)1/1, chroma-site=(string)jpeg, colorimetry=(string)bt601, framerate=(fraction)23/1"

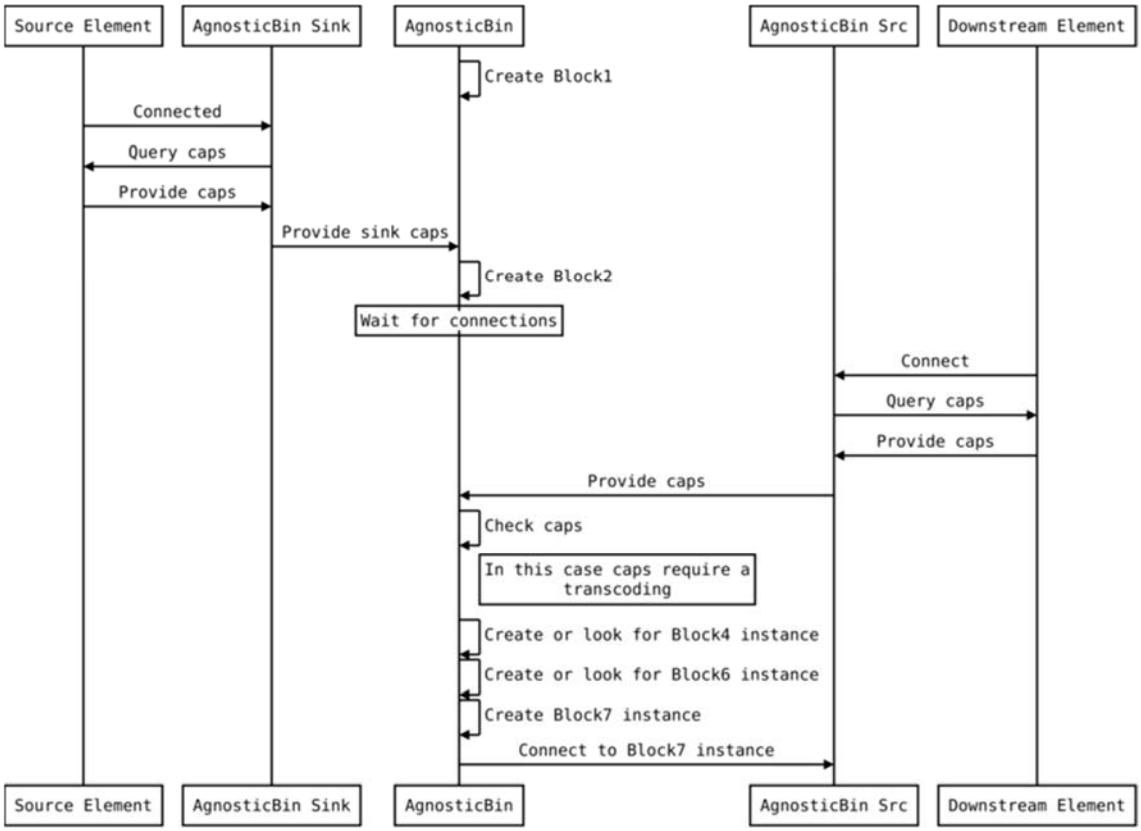

Figure 22. This diagram shows the process executed by agnostic bin to perform its internal connections and blocks construction.

In this case the capabilities requested by downstream element are raw and a decoding of the input media should be done. First, the element that feeds media to agnosticbin (Source Element) needs to be connected and provide the capabilities of the media. This will create Blocks 1 and 2. Then agnosticbin waits for downstream elements to feed media to. When a new element is connected, their caps are queried in order to know the format that that agnostic should be feed to it, in this case the requested media is raw. If previously Block4 has been created, it is used directly, otherwise it is created. Then, a Block5 is created and connected to provide media to downstream element. In this specific example, the caps presented by the Source Element are represented by the following string "video/x-vp8, profile=0, streamheader=< 4f5650383001010003... > ,width=(int)854,height=(int)480,pixel-aspect-ratio=1/1,framerate=23/1". The caps requested by the Downstream Element are "video/x-h264" and the resulting final caps provided by the AgnosticBin are "video/x-h264, stream-format=(string)avc, alignment=(string)au,profile=(string)baseline, codec_data=(buffer) 0142c01e0101000f6742c01e8c8d406c1eb3700f08846a01000468ce3c80, width=(int)854, height=(int)480, pixel-aspect-ratio=(fraction)1/1, framerate=(fraction)23/1"

4.5.5.3 Hubs

4.5.5.3.1 KmsBaseHub

The base class for all hubs is the abstract class KmsBaseHub, which provides its descendants the ability to connect and disconnect HubPorts for media transmission. The following diagram shows the inheritance model, and the different types of specific hub classes that exist. As it can be observed, hubs are GStreamer bins, but they do not inherit from KmsElement as other media capabilities introduced above. This is due to the fact that KmsElements have only two sinks (one for audio and one for video) while hubs need to be capable of having variable on-demand number of sinks.

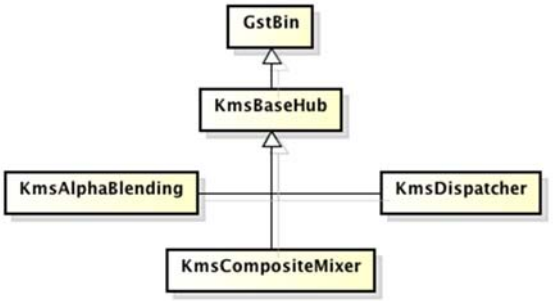

Figure 23: KmsBaseHub UML diagram
Hubs are GStreamer bins which do not inherit from KmsElement because they have variable on-demand number of sinks.

Hubs send and receive media through HubPorts, which inherits from 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.

4.5.5.3.2 KmsHubPort

KmsHubPort is a specification of KmsElement that is created to be connected to a Hub. These elements perform a media adaptation for each output using agnosticbin and offer a common way to connect hubs to other elements. Additionally, they help the hub to know what inputs are associates with what outputs so the hub can do special actions if required.

4.5.5.3.3 KmsCompositeMixer

KmsCompositeMixer is a hub that allows combining several input video streams in one single output stream through a composite layout. This layout is divided in two columns, and has as many rows as needed depending on the number incoming video streams. Thus, if there are eight input videos, the output stream will have two columns and four rows. All videos will have the same size in the grid, even if their original size is different. The streams will appear in the grid from left to right, top to bottom in order of connection. In turn, incoming audio streams are mixed together following a simple non-weighed sum mechanism. For avoiding echoes, the output audio offered to one participant does not include its own audio stream into the mix.

One of the most typical uses of KmsCompositeMixer is to save bandwidth in a multiconference scenario. For example, imagine that you have 10 users in a videoconferencing meeting. Without a mixer, each user would be emitting its 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 routing infrastructure. This may bring bottlenecks to the media infrastructure itself, but most important, to the client devices. For example, mobile devices are typically connected through 3G/4G networks, which may run into problems for supporting the 10 video streams required in our use case. In addition, this may also impact on use costs for participating in the call. With a composite mixer, we can combine all the streams into one single feed, each user can have only one incoming media feed, having a total of only 2 connections per user.

Figure 25 represents the internal structure of the KmsCompositeMixer. The grey boxes outside the main 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. At is can be observed, at the heart of the KmsCompositeMixer we have to GStreamer bins:

- The GstCompositor, which performs the composition of the video frames in raw. This means that KmsCompositeMixer requires the decoding of each of the incoming video streams for perform on them a scaling (GstVideoScale) plus the final composition mixing.
- KmsAudioMixer, which is a bin implementing the audio mixing following the scheme described above.

4.5.5.3.4 KmsDispatcher and KmsDispatcherOneToMany

KmsDispatcher and KmsDispatcherOneToMany are hubs that perform just a media routing operation without any kind of processing and without requiring any coding operations. The KmsDispatcher has the ability of routing from N inputs to M outputs. Hence, every HubPort can obtain at its source any of the media streams received from any of the rest of HubPort sinks. KmsDispatcherOneToMany is simpler because it just clones the stream at its master HubPort sink into all the sources of the connected HubPorts. The master HubPort can be switched at wish using that hub API. As a last remark, media routing is performed in both audio and video, so that the audio and video streams entering into a dispatcher are routed together towards its destination ports.

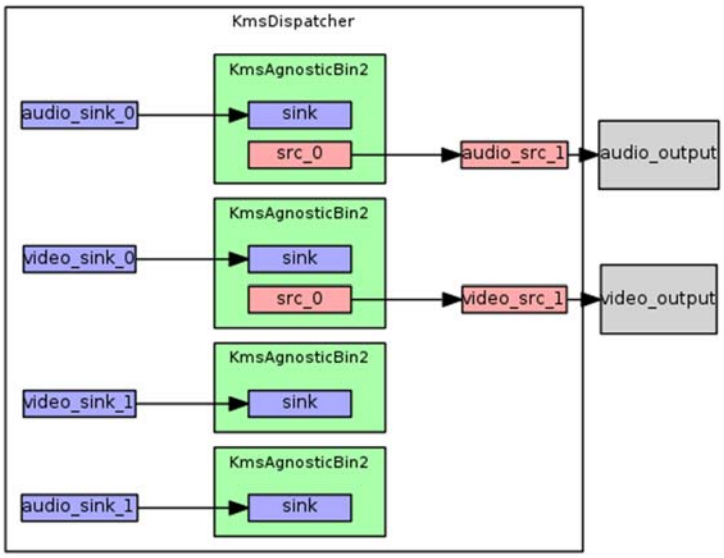

Figure 24: Dispatcher internal structure

The internal structure of a KmsDispatcher is shown in Figure 24. As it can be observed, the KmsDispatcher just contains a switching matrix connecting sinks and sources through AgnosticBin. This somehow means that both KmsDispatcher and KmsDispatcherOneToMany are compact ways of interconnecting media elements, but the functionality exposed by dispatchers can be accessed by directly interconnecting media elements.

4.5.5.3.5 KmsAlphaBlending

This type of hub allows blending several input video streams over a master background video. The blending operation works in layers and fully honors pixel transparency (i.e. alpha channel), which means that areas at the lower layers may be visible in the blended stream as long as all upper layer are transparent in that area. The master stream, which is set through specific API primitives, defines the output size, and the other streams should adapt, according to their configuration, to that size. The internal structure of the

KmsAlthaBlending is represented in Figure 26. As in the Composite, the grey boxes outside of the KmsAlphaBlending box represent different HubPorts connected to it.

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.

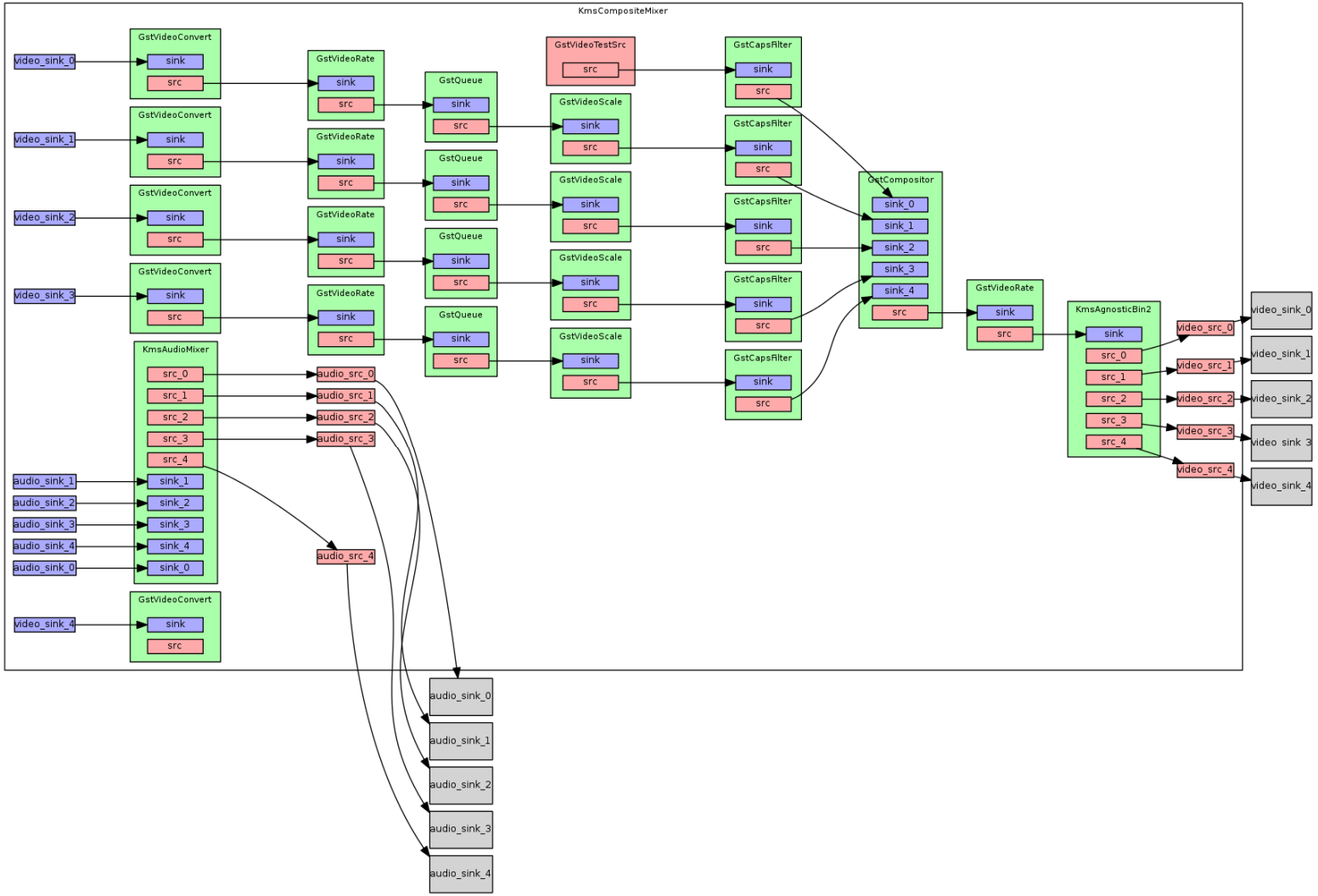

Figure 25: KmsComposteMixer internal structure

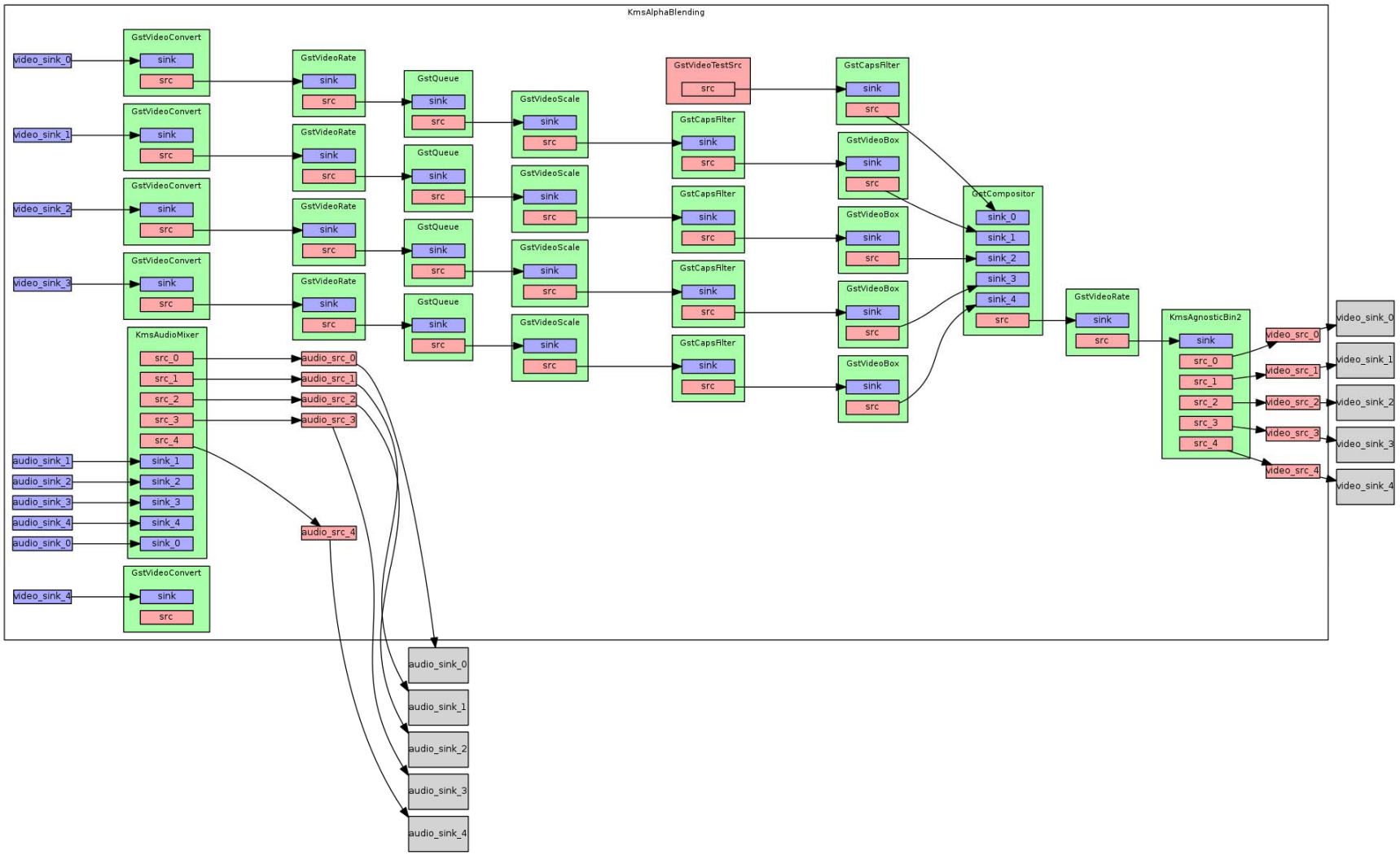

Figure 26: Alpha Blending internal structure

4.5.5.4 The RTP Stack

As a RTC media server, KMS provides a complete RTP stack enabling a rich set of features for real-time communications which include SDP negotiation, SRTP, DTLS, AVPF, WebRTC, etc. All these capabilities are implemented as KmsElements derivate classes following the inheritance hierarchy shown in Figure 27.

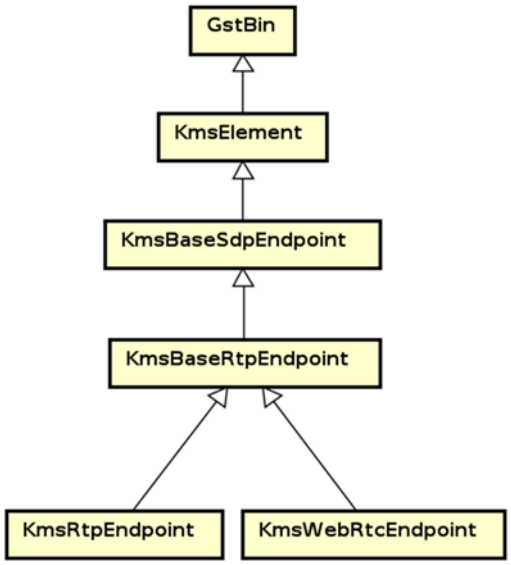

Figure 27: WebRTC and RTP endpoints UML diagram

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

4.5.5.4.1 KmsBaseSdpEndpoint

This class manages media sessions negotiated through the Session Description Protocol (SDP) [RFC4566]. When initiating multimedia teleconferences, voice-over-IP calls, streaming video, or other sessions, there is a requirement to convey media details, transport addresses, and other session description metadata to the participants. SDP provides a standard representation for such information, irrespective of how that information is transported. SDP is purely a format for session description -- it does not incorporate a transport protocol, and it is intended to use different transport protocols as appropriate, including the Session Announcement Protocol, Session Initiation Protocol, Real Time Streaming Protocol, electronic mail using the MIME extensions, and the Hypertext Transport Protocol.

The SDP negotiation process involves two steps:

- SDP offer generation, which is in fact a declaration of the own capabilities of the offerer. This may include information such as where can the media be made available (ip address and port), which formats and codecs can be used, etc. The SDP offer 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
- SDP answer. Once the answer is generated, it is sent back to the original sender. This SDP answer is in fact the “how and where” of the media exchange.

In order to perform this SDP negotiation capabilities, KmsBaseSdpEndpoint is associated to a KmsSdpAgent object, which carries out the SDP negotiation logic implementing the SDP RFC and some extensions to support AVP, AVPF, SAVP, SAVPF, etc. These extensions are provided by each KmsBaseSdpEndpoint subclasses

adding KmsSdpHandler intances (example: the KmsRtpEndpoint uses an AvpMediaHandler, while the KmsWebRtcEndpoint uses a SavpfMediaHandler)

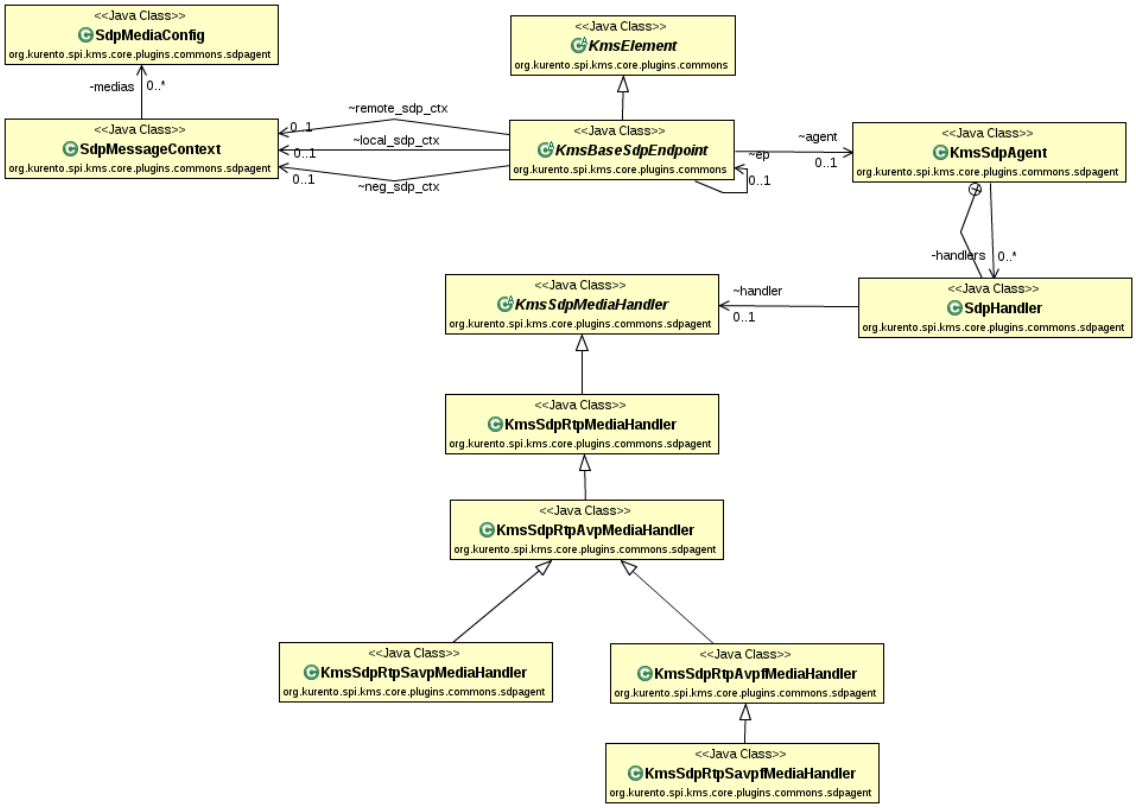

Figure 28: KmsBaseSdpEndpoint class diagram

Moreover, KmsBaseSdpEndpoint stores the local, remote and negotiated SDPs to be used by each subclass to configure themselves.
In order to facilitate the management of these SDPs, they are structured using the SdpMessageContext (for the general settings of the session) and the SdpMediaConfig (for each audio or video media settings) abstractions.

4.5.5.4.2 KmsBaseRtpEndpoint

This class encapsulates media in Real-Time Transport Protocol (RTP) [RFC3550] 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. In addition, this class provides an implementation for the AVP [RFC3551] and AVPF [RFC4585] RTP profiles.

The KmsBaseRtpEndpoint internal structure is depicted in Figure 29. This structure manages RTP session, which consists of an IP address and a pair of ports for RTP and RTCP for each multimedia stream through the use of the GstRtpBin GStreamer bin. This bin allows synchronizing audio and video and provides a jitterbuffer implementation capable of managing packet loss, packet reordering and network jitter. The GstRtpBin also provide SSRC (de)multiplexing capabilities inside sessions. The KmsBaseRtpEndpoint also provides RTP (de)payloading capabilities transforming media buffers and RTP packets among each other in alignment with the appropriate payloading recommendations (e.g. RFC6184, RFC4867, etc). As it can be observed,

RTP and RTCP video and audio streams are managed separately on the bin through two independent pathways.

The KmsBaseRtpEndpoint endpoint is transport agnostic, delegating it to its subclasses. One of the most relevant features offered to subclasses, apart of course of the ability to send and receive media in real time, is to do so with the best possible QoS made available by the network, especially in what concerns to bandwidth. Thus, if an application is trying to send video of a higher quality than what the network is able to transmit (for instance, trying to send FullHD video over a 3G connection), the connection will become congested, packages will be lost and the receiving end could eventually stop viewing any 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.

4.5.5.4.3 KmsRtpEndpoint

This endpoint manages RTP connections using UDP as transport. As shown in Figure 27, it inherits form BaseRtpEndpoint and BaseSdpEndpoint, so that it inherits their capabilities. As a result, this class only requires implementing its specific transport. The internals of a KmsRtpEndpoint are depicted in Figure 30. As it can be observed, the GstRtpBin and its associated capabilities are inherited from its ancestors. The only additional elements are related to transport, which is physically implemented through the GstUdpSrc and GstUpdSink GStreamer elements.

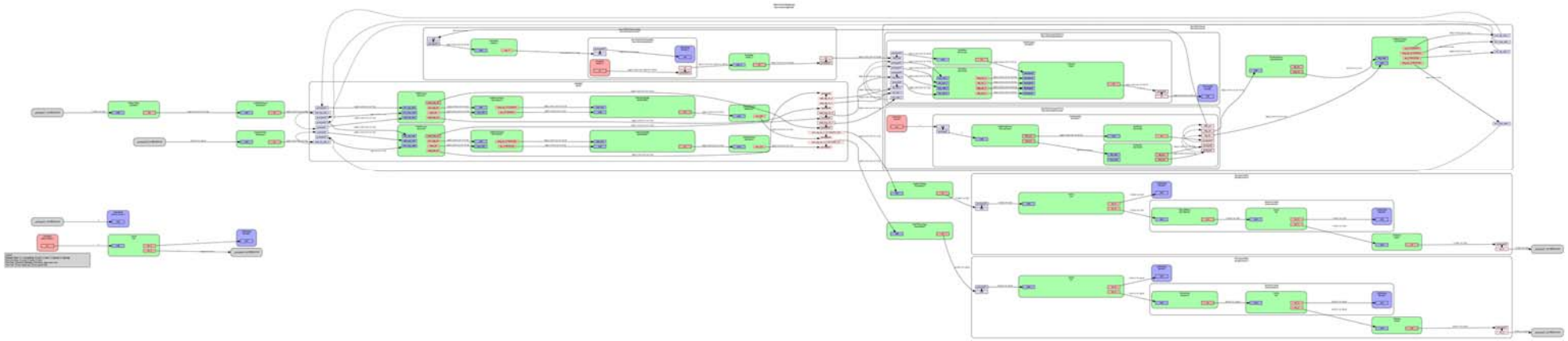

Figure 29: BaseRtpEndpoint internal structure

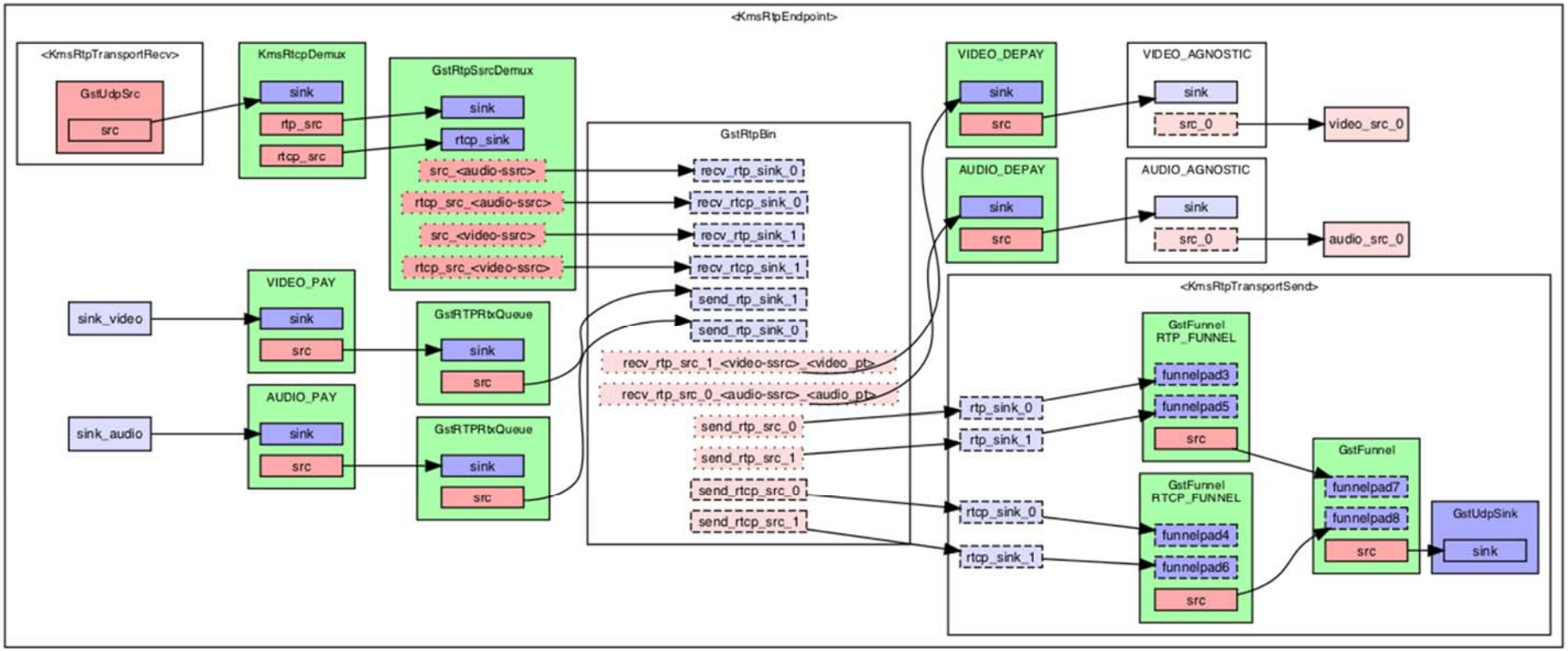

Figure 30: RtpEndpoint internal structure

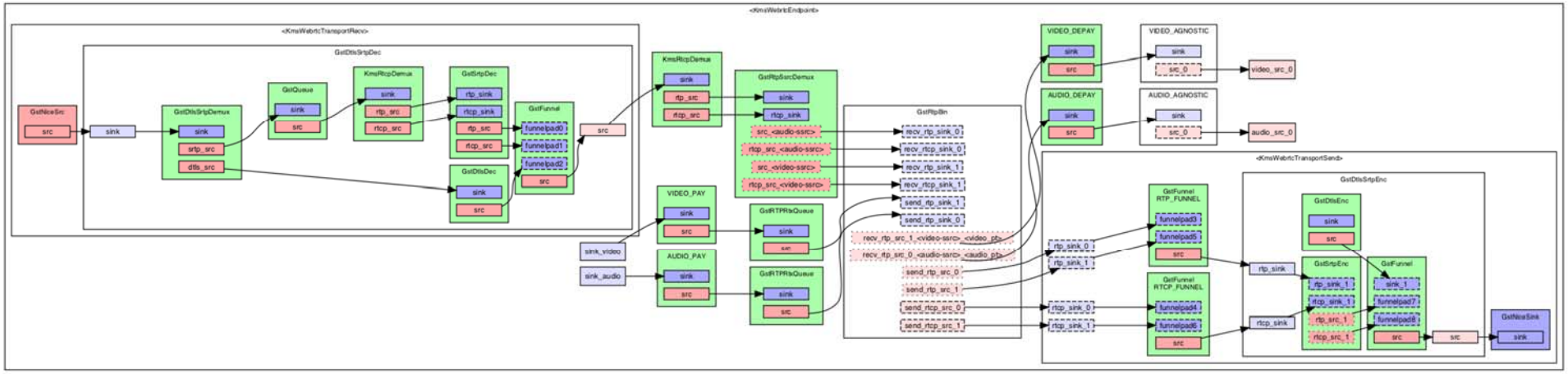

Figure 31: WebRtcEndpoint internal structure

4.5.5.4.4 KmsWebRtcEndpoint

This is one of the most relevant media capabilities of KMS. It provides the ability to the media server to communicate media using WebRTC standards. Following standard terminology [WEBRTCOV], the KmsWebRtcEndpoint is a non-browser WebRTC endpoint (also known as WebRTC device), meaning that it provides full compatibility with WebRTC transport protocols but it does not provide the standard W3C JavaScript APIs. For this, the KmsWebRtcEndpoint 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. Their role can be seen in the internal schema of KmsWebRtcEndpoint shown in Figure 31, along with all the rest of GStreamer elements from the endpoint. The box on the left represents the decoding element, and is connected with the sink pad of the endpoint, while the encoding element is located on the right-most side of the image, connected to the source pad of the element. For the generation of the keys, a certificate is needed. Developers can provide it to the KmsWebRtcEndpoint, though it also has the capability of generating the certificate if none is provided.

SRTP packets are exchanged through an Interactive Connectivity Establishment (ICE) connection. ICE comprise a set of NAT traversal techniques 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 are be sent to those candidates, unless a better candidate is discovered during the media exchange. For this purpose, two GStreamer-based 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. This is a feature called DataChannels, which is also supported. This feature is implemented using a Session Control Transmission Protocol (SCTP) connection encrypted using DTLS. In consonance with the existing structure of the KmsWebRtcEndpoint, the approach is to create two GStreamer elements (KmsSctpDec and KmsSctpEnc) that marshall and unmarshall SCTP packets. These two elements are then connected to the existing GstDtlsSrtpDec and GstDtlsSrtpEnc, where the packets are encrypted or decrypted. The data received from the network is send by data stream defined in MediaElement class. The data that comes from other MediaElements through the data stream is send using the WebRTC data channels mechanism.

Considering the whole inheritance hierarchy, the KmsWebRtcEndpoint supports the following standards and drafts:

- RFC 0778, User Datagram Protocol
- RFC 3550, RTP: A Transport Protocol for Real-Time Applications.
- RFC 3551, RTP Profile for Audio and Video Conferences with Minimal Control
- RFC 3711, The Secure Real-time Transport Protocol (SRTP)

- 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 5104, Codec Control Messages in the RTP Audio-Visual Profile
- RFC 5124, Extended Secure RTP Profile for Real-time Transport Control Protocol (RTCP)-Based Feedback (RTP/SAVPF)
- RFC 5245, Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols
- RFC 5398, Session Traversal Utilities for NAT (STUN)
- RFC 5285, A General Mechanism for RTP Header Extensions
- RFC 5506, Support for Reduced-Size Real-Time Transport Control Protocol (RTCP): Opportunities and Consequences.
- 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
- RFC 5777, Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN).
- RFC 6184, RTP Payload Format for H.264 Video

Drafts:

- draft-ietf-rtcweb-data-channel, WebRTC Data Channels
- draft-ietf-rtcweb-data-protocol, WebRTC Data Channel Establishment Protocol
- draft-ietf-rtcweb-jsep, Javascript Session Establishment Protocol
- draft-ietf-rtcweb-rtp-usage, Web Real-Time Communication (WebRTC): Media Transport and Use of RTP
- draft-lennox-avtcore-rtp-multi-stream, Sending Multiple Media Streams in a Single RTP Session
- draft-ietf-mmusic-sdp-bundle-negotiation, Negotiating Media Multiplexing Using the Session Description Protocol (SDP)
- raft-alvestrand-rmcat-congestion, A Google Congestion Control Algorithm for Real-Time Communication
- draft-alvestrand-rmcat-remb, RTCP message for Receiver Estimated Maximum Bitrate
- abs-send-time, The Absolute Send Time extension
- draft-ietf-mmusic-trickle-ice, Tricke ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol
- draft-uberti-rtcweb-plan. Plan B: a proposal for signaling multiple media sources in WebRTC
- draft-roach-mmusic-unified-plan, A Unified Plan for Using SDP with Large Number of Media Flows
- draft-ietf-payload-vp8, RTP Payload Format for VP8 Video
- draft-ietf-payload-rtp-opus, RTP Payload Format for the Opus Speech and Audio codec

4.5.5.5 URI Endpoints

4.5.5.5.1 KmsPlayerEndpoint

This type of endpoint plays videos accessible from a URI (files, online resources, etc.) Supported URI schemes include:

- File URIs (starting with file:///)
- HTTP URIs (starting with http://)
- RTSP URIs (starting with rtsp://)

It is capable of playing most types of media formats and codecs, as supported by the GStreamer element Uridecodebin, which gives to this endpoint the capability of reading media streams for URIs. Among the supported containers we can find *webm*, *mp4*, *avi*, *mkv*, and *raw*. Among the codecs H.264, H.263, VP8, VP9, AMR, Opus, Speech, MP3 and PCM. Figure 32 shows the inheritance structure of this endpoint, which comprises a KmsElement and a KmsUriEndpoint as ancestors.

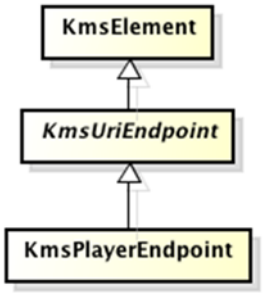

Figure 32: KmsPlayerEndpoint UML diagram

Figure 33 shows KmsPlayerEndpoint internal structure. As it can be observed, the Uridecodebin reads media from the specified URIs and provides it to GstAppSink elements, which in turn forward the media to the corresponding KmsAgnosticBins (inherited from KmsElement) suitable for serving the media to other media elements downstream.

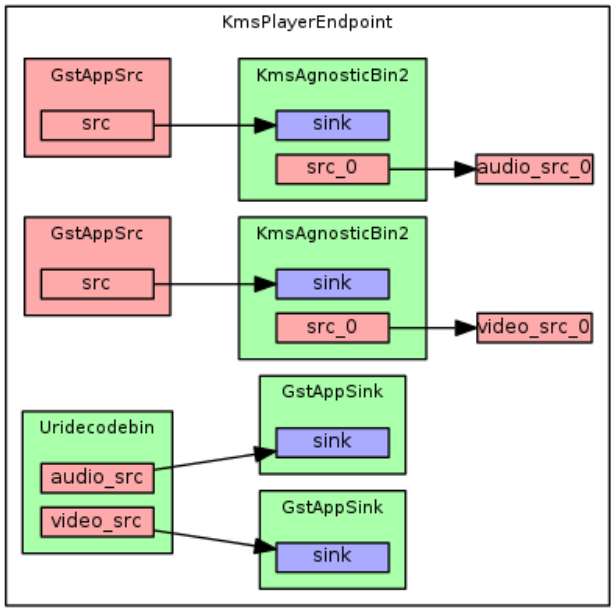

Figure 33: KmsPlayerEndpoint internal structure

4.5.5.5.2 KmsRecorderEndpoint

This endpoint is in charge of storing media streams in a designated location, accessible through a URI. The supported URI schemes are:

- File URIs (starting with file:///)
- HTTP URIs (starting with http:///)

File URIs record media resources in the specified file in the local file system. HTTP URIs forward media contents through HTTP POST methods following this scheme:

The POST requests are sent using a chunked encoding. This HTTP header is included Transfer-Encoding: chunked

KmsRecorderEndpoint supports the recording of files in different formats, including *mp4* and *webm*. The following restrictions apply to these formats:

Restrictions:

- When only audio or video is going to be sent, the correct format should be used (WEBM_AUDIO_ONLY, WEBM_VIDEO_ONLY, etc), otherwise the media will be buffered in memory until the recorder is stopped.
- When MP4 format if used transcodifications may be required if recorder input is switched even if the source is already in H264, this is because the file format exiges the codec to be exactly the same across the whole file.
- Webm container stores audio in opus format. This is done to improve the quality of the recording; bug may cause problems with some players.

Figure 34 shows the inheritance structure of KmsRecorderEndpoint.

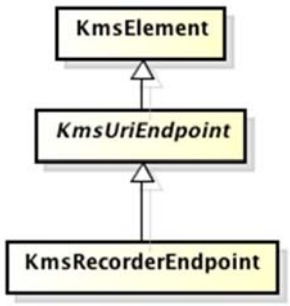

Figure 34: KmsRecorderEndpoint UML diagram

Figure 35 shows the internal structure of KmsRecorderEndpoint. As it can be observed, it leverages GStreamer components for performing the appropriate encodings and archiving operations.

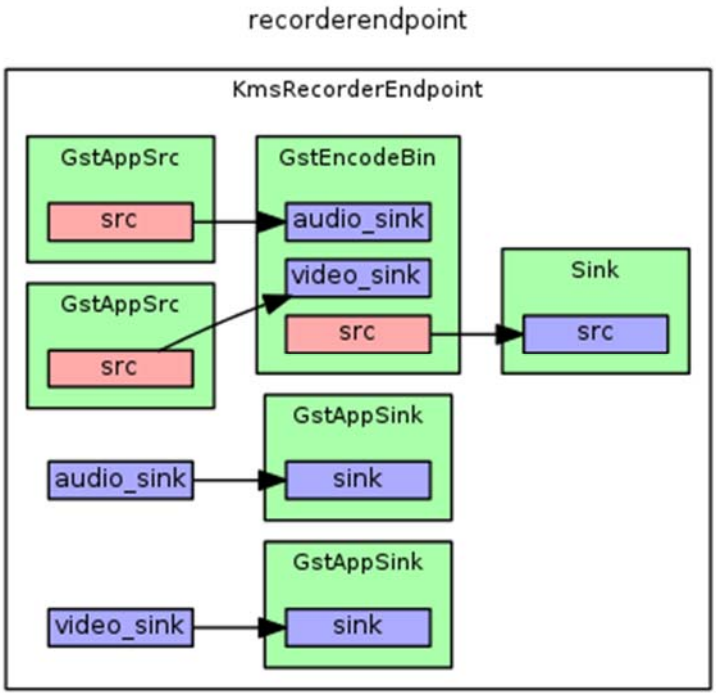

Figure 35: KmsRecorderEndpoint internal structure

4.5.5.5.3 KmsHttpEndpoint

KmsHttpEndpoint is the parent class for all HTTP endpoints. HTTP endpoints are elements that send or receive media through the HTTP protocol leveraging HTTP methods such as GET, POST or PUT. KmsHttpEndpoint provides some common HTTP functionalities for all such endpoints.

As shown in Figure 36, at the time of this writing the media server only implements a specific HTTP endpoint basing on POST messaged. Further HTTP endpoints might be added in the future.

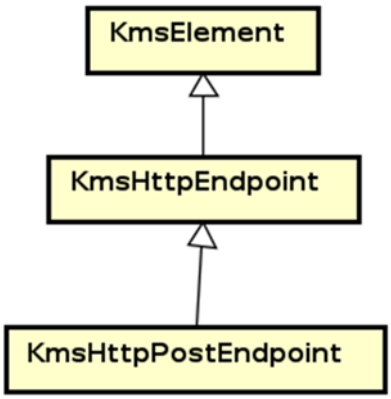

Figure 36: KmsHttpEndpoint UML diagram

4.5.5.5.4 KmsHttpPostEndpoint

HttpPostEndpoint is a source element, just like PlayerEndpoint but instead of getting media from a URI, it offers an HTTP server where the clients can send media using POST requests.

POST requests can be a single request where all the body has the content or can be multipart (as for example while uploading a file from a web form).

When the connection with the endpoint is closed and the session expires, internal pipeline emits an EOS to notify that the file upload has finished.

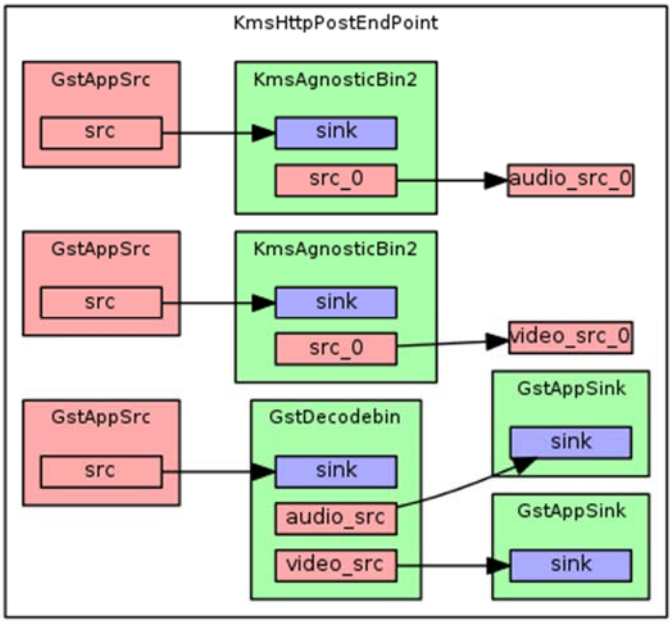

Figure 37: KmsHttpPostEndpoint internal structure

4.5.5.6 Filters

4.5.5.6.1 KmsFaceDetectorFilter

The KmsFaceDetectorFilter is a filter able to detect faces in a frame. This filter is based on the Haar Cascade Detector implemented by OpenCV. It is using the face training provided by OpenCV. This filter sends an event with info about detected faces (position and size) packaged as a GStreamer event.

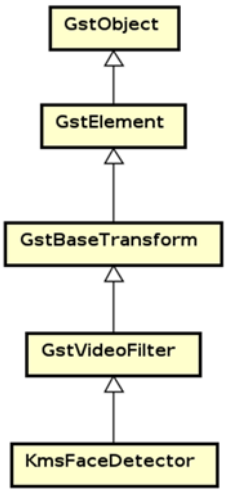

Figure 38: KmsFaceDetectorFilter internal structure

As shown in Figure 38, this filter implements the interface provided by GstVideoFilter class of GStreamer.

4.5.5.6.2 KmsImageOverlayFilter

This element can read events from KmsFaceDetectorFilter and draw a preloaded image over the detected faces in the media. The overlaid image can be read from a file or from an http url.

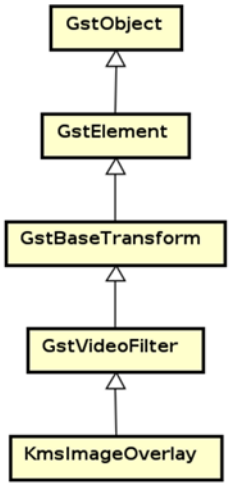

Figure 39: KmsImageOverlayFilter internal structure

As shown in Figure 39, the KmsIageOverlaFilter implements the interface provided by GstVideoFilter class of GStreamer.

4.5.5.6.3 KmsFaceOverlayFilter

The KmsFaceOverlay Filter is a GStreamer bin (see Figure 40) that combines the KmsFaceDetectorFilter and the KmsImageOverlayFilter to create a filter able to detect and draw faces in a flow.

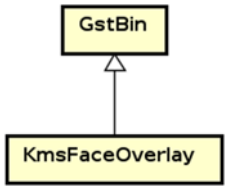

Figure 40: KmsFaceOverlayFilter internal structure

4.5.5.6.4 KmsLogoOverlayFilter

This filter is able to draw a predefined image over the video. It is possible to configure the size and the position of the overlaid image. This image can be loaded from a file or from an HTTP URI.

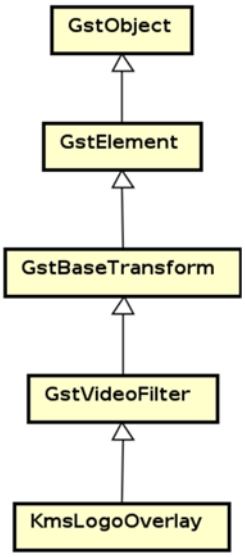

Figure 41: KmsLogoOverlayFilter internal structure

4.5.5.6.5 KmsOpenCVFilter

KmsOpenCVFilter is closely related with generic OpenCvFilter c++ class. KmsOpenCVFilters converts gstreamer buffer to opencv Mat data types. This Mat structure is passed to C++ class that is passed to the plugins implementor code to be processed. Thanks to this element, opencv programmers do not need to know anything about gstreamer but only about opencv.

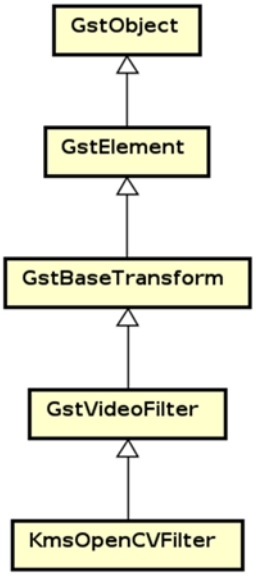

Figure 42: KmsOpenCVFilter internal structure

4.5.5.6.6 ZbarFilter

The ZBarFilter detects barcodes and retrieves info about them. It is implemented based on GstVideoFilter class (see Figure 43).

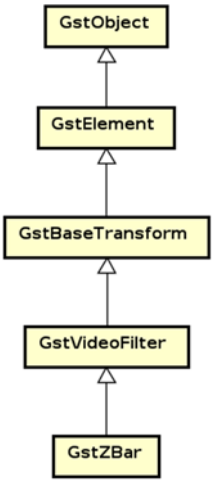

Figure 43: ZBarFilter internal structure

4.6 Media Server modules

The Media Server architecture is based on pluggable modules. This means that all media capability types (i.e. Media Objects) are shipped as modules and that those modules may be loaded upon Media Server startup. Each module provides the ability of instantiating and managing one or several media capabilities (i.e. Media Element types).

A module is a dynamic library (.so files in UNIX systems) that is loaded in runtime, this library has a special symbols with the information of the module, this symbols are:

getModuleDescriptor that returns the kmd that generated this module, getModuleName that returns the module name, getModuleVersion that return the version of the module and getFactoryRegistrar that returns a FactoryRegistrar with has a list of available factories associated to their names. In practice, this means that, for every Media Object class that can be managed by the Media Server Manager; the corresponding factory must be provided by a loaded module. Hence, the module provides just the logic needed for creating Media Objects, which in turn have the logic for creating and managing their associated media capabilities

The Media Server is shipped with a number of built-in modules, which were introduced in sections above. In addition, the Media Server accepts custom modules, which extend Media Server capabilities and which can be created by developers and deployed on Media Server start-up without requiring any kind of Media Server code re-compilation. The following sections are devoted to explaining the Media Server module system.

4.6.1 Built-in modules

These modules are provided off-the-self by the Media Server (i.e. its code is shipped with the Media Server), so that the Media Server cannot be started without loading all of them. For a number of practical reasons, built-in modules have been organized in three groups:

- **kms-core:** This module provides all the abstract classes that are used by the specific media capabilities. These include MediaObject, Hub, Filter, SdpEndpoint, etc. In addition, this module provides several special Media Objects such as MediaPipeline, ServerManager or HubPort.
- **kms-elements:** These comprise a number of modules implementing specific Media Element features. In particular, all specific Media Server Endpoints and Hubs belong to this category. Among these modules we can find the WebRtcEndpoint, the RtpEndpoint, the Composite, etc.
- **kms-filters:** This contains implementations of all the built-in filters, which include the FaceOverlayFilter, the GStreamerFilter or the ImageOverlayFilter.

4.6.2 Custom modules

Custom modules have the same architecture than built-in modules. There are two differences between custom modules and built-in modules:

1. KMS has building dependencies for built-in modules.
2. The custom modules have building dependencies of built-in modules.

4.6.2.1 Creating custom modules

It is possible to create any kind of MediaElement as a custom module (endpoints, filters, passthrough elements, etc.) We provide a tool called kurento-module-scaffold ready to generate the structure of filters as custom modules, but can be easily modified in order to implement other types of MediaElements. Two types of custom filters can be developed using the scaffold tool:

- Modules based on OpenCV. These kinds of modules are recommended for developing computer vision filters. In this kind of filters, the developer only needs to add C++ code with the OpenCV logic into a function ready accept it.
- Modules based on GStreamer. These kinds of modules are more powerful but also are more difficult to develop. The developer should develop a GStreamer plugin and add C++ code in the server files to handle the properties of that plugin.

The tool usage is different depending on the chosen flavor.

OpenCV module:

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

Gstreamer module:

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

The tool generates the folder tree, all the necessary CmakeLists.txt files 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 ..
```

In a OpenCV module there are 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 the 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 the filter logic should be implemented inside this function.

In the other hand, the Gstreamer filters have two directories inside the src folder: the gst-plugins folder containing the implementation of the GStreamer plugin (the kurento-module-scaffold generates a dummy filter but it should be changed by the filter logic). Inside the server/objects folder you have two files:

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

From the code into ModuleNameImpl.cpp is possible to invoke methods of the GStreamer element but all the module logic (including media processing) will be implemented in the GStreamer element.

Once the module logic is implemented and the compilation process is finished, it is necessary to install the custom module in the system in order to KMS to load the module in runtime. The best way to do it is to generate the Debian package (debuild -us -uc) and install it (dpkg -i <package_name>).

It is possible to generate client code (Java or JavaScript) for custom modules.

- Java code (execute from build directory):
  - cmake .. -DGENERATE_JAVA_CLIENT_PROJECT=TRUE
  - make java_install
- Javascript module (execute from build directory):

- cmake .. -DGENERATE_JS_CLIENT_PROJECT=TRUE
- JavaScript code will be available in a new js directory.

4.7 Kurento Module Description language

The KMD (Kurento Module Descriptor is the interface description language (IDL) of Kurento. These files allow defining the capabilities of a module and generating code for the server side and the client side. KMD files define: classes, complex types (registers and enums), events, and constructors.
KMD supports several data types:

- int
- boolean
- float
- double
- String
- [ ] (Arrays) (int [], boolean[], …)
- <> (Map <String, type>) (int <>, boolean <>, String <>, …)
- ComplexTypes defined by the user.

A KMD file is defined using JSON and contains the following fields:

- remoteClasses
- complexTypes
- events

```
{
  "remoteClasses": [
    {
      "name": "Example1",
      "extends": "Filter",
      "doc": "Example1 interface. Documentation about the module",
      "constructor": {
        "doc": "Create an element",
        "params": [
          {
            "name": "mediaPipeline",
            "doc": "the parent :rom:cls:`MediaPipeline`",
            "type": "MediaPipeline",
            "final": true
          }
        ]
      }
    }
  ]
}
```

Figure 44: KMD Filter Example (I)

The properties generate setters and getters methods in server and APIs code. A property is defined by the following attributes:

- name (mandatory)
- doc (mandatory)
- type (mandatory)
- final
- readOnly

```
{
  "remoteClasses": [
    {
      "name": "Example2",
      "extends": "Filter",
      "doc": "Example2 interface. Documentation about the module",
      "constructor": {
        "doc": "Create an element",
        "params": [
          {
            "name": "mediaPipeline",
            "doc": "the parent :rom:cls:`MediaPipeline`",
            "type": "MediaPipeline",
            "final": true
          }
        ]
      },
      "properties": [
        {
          "name": "isLive",
          "doc": "Define is the source is live",
          "type": "boolean",
          "final": true
        },
        {
          "name": "volume",
          "doc": "Get the media volume",
          "type": "double",
          "readOnly": true
        },
        {
          "name": "mute",
          "doc": "Audio mute",
          "type": "boolean"
        }
      ]
    }
  ]
}
```

Figure 45: KMD Filter Example (II)

Methods generate a method with the same name in server and APIs code. A method is defined with the following attributes:

- name (mandatory)
- doc (mandatory)
- params (mandatory. It can be an empty array)
  - name (mandatory)
  - doc (mandatory)
  - type (mandatory)
  - optional (to indicate if param is required or not)
- return
  - doc (mandatory if return is defined)
  - type (mandatory if return is defined)

```
{
  "remoteClasses": [
    {
      "name": "Example3",
      "extends": "Filter",
      "doc": "Example3 interface. Documentation about the module",
      "constructor": {
        "doc": "Create an element",
        "params": [
          {
            "name": "mediaPipeline",
            "doc": "the parent :rom:cls:`MediaPipeline`",
            "type": "MediaPipeline",
            "final": true
          }
        ]
      },
      "methods": [
        {
          "name": "increaseIntensity",
          "doc": "Increase the intensity 0.1",
          "params": []
        },
        {
          "name": "increaseDelay",
          "doc": "Increase the echo delay. Returns the new delay in ms",
          "params": [
            {
              "name": "ms",
              "doc": "ms to increase the delay",
              "type": "int"
            }
          ],
          "return": {
            "doc": "The new delay",
            "type": "int"
          }
        }
      ]
    }
  ]
}
```

Figure 46: KMD Filter Example (III)

Events generate a class with event properties in server and APIs code. They also generate a signal to throw events from the server and listeners in APIs code. An event is defined with the following attributes:

- name (mandatory)
- extends
- doc (mandatory)
- properties (mandatory. It can be an empty array)
    - name (mandatory)
    - doc (mandatory)
    - type (mandatory)

```
{
  "remoteClasses": [
    {
      "name": "Example4",
      "extends": "Filter",
      "doc": "Example4 interface. Documentation about the module",
      "constructor": {
        "doc": "Create an element",
        "params": [
          {
            "name": "mediaPipeline",
            "doc": "the parent :rom:cls:`MediaPipeline`",
            "type": "MediaPipeline",
            "final": true
          }
        ]
      },
      "events": [
        "AudioAnalisis"
      ]
    }
  ],
  "events": [
    {
      "name": "AudioAnalisis",
      "extends": "Media",
      "doc": "Info related with audio.",
      "properties": [
        {
          "name": "time",
          "doc": "",
          "type": "int"
        },
        {
          "name": "duration",
          "doc": "",
          "type": "int"
        },
        {
          "name": "rglevel",
          "doc": "",
          "type": "double"
        }
      ]
    }
  ]
}
```

Figure 47: KMD Filter Example (IV)

KMD files allow defining complex types to be used as parameter or return values. A complex type is defined with the following attributes:

- name (mandatory).
- extends.
- Type Format (mandatory): REGISTER or ENUM.
- doc (mandatory)
- properties:
    - name (mandatory)
    - doc (mandatory)
    - type (mandatory)

```
{
  "complexTypes": [
    {
      "name": "Point",
      "doc": "Description of the complexType",
      "typeFormat": "REGISTER",
      "extends": "Media",
      "properties": [
        {
          "name": "x",
          "doc": "x coordinate",
          "type": "int"
        },
        {
          "name": "y",
          "doc": "y coordinate",
          "type": "int"
        }
      ]
    }
  ]
}
```

Figure 48: KMD Filter Example (V)

4.8 Information for developers

The NUBOMEDIA Media Server has been implemented as a set of evolutions and extensions on top of Kurento Media Server. This means that, for the purpose of this document, NUBOMEDIA Media Server and Kurento Media Server are the same thing.

Kurento Media Server relevant developers' information is specified in the following sections.

4.8.1 License

All Kurento Media Server software artifacts have been released under the terms and conditions of Apache 2.0 license.

4.8.2 Documentation

Kurento media server documentation is available at the Kurento documentation repository:

- http://doc-kurento.readthedocs.org/en/latest/

4.8.3 Source code repositories

Kurento Media Server software artifacts are distributed under the following repository structure:

kms-elements

- Description: This repository contains the implementation of all specific built-in media elements. This includes all endpoints.
- URL: https://github.com/nubomedia/kms-elements

kms-core

- Description: This repository contains the implementation of all abstract media elements.
- URL: https://github.com/nubomedia/kms-core

kms-filters

- Description: This repository contains the implementation of all built-in filters.
- URL: https://github.com/nubomedia/kms-filters

kms-jsonrpc

- Description: This repository contains the implementation of the Kurento Protocol.
- URL: https://github.com/nubomedia/kms-jsonrpc

kurento-media-server

- Description: This class contains the media server executable as well as all the Media Server Manager modules involved in communication and control.
- URL: https://github.com/nubomedia/kurento-media-server

kurento-module-creator

- Description: this project contains all the tools required for creating Kurento modules.
- URL: https://github.com/nubomedia/kurento-module-creator

5 NUBOMEDIA Custom Modules

5.1 Chroma

5.1.1 Rationale

The chroma key is a technique used heavily to remove or replace the background of a video. It is a technique used in cinema and television production. In most cases, the background used is green to enable easier the detection and replacement of the background. In this case, this filter can be used to replace a green background by any preloaded image in real time.

5.1.2 Objectives

In order to develop this module, several objectives were necessary:

- To develop a GStreamer filter able to detect the target color in the image.
- To enable the chroma to be based in any background color.
- To get the image to replace the background.
- To expose the capabilities through the API.

5.1.3 Software architecture

The Chroma module has been implemented based on the GstVideoFilter of GStreamer. The access to image pixels and their replace have been done using the OpenCV library. If the background image is a url, the module uses the SOUP HTTP library to download the image.

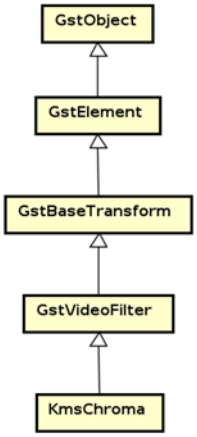

Figure 49: KmsChroma internal structure

5.1.4 Capabilities

Filter capabilities:

- Set the background color in an automatic way. The filter calculates the background color with the color in a small region, configured in the constructor of the filter, in the first five seconds of video.
- Set the background image from an API function (setBackground) in any moment.
- Unset the background image from an API function (unsetBackground) in any moment.
- Replace a pixel in the image with the same pixel in the background image if the pixel fit into the configured background color.

5.1.5 Information for developers

The chroma module has been released under the terms and conditions of the LGPL v2.1 license. The source code can be obtained in the following repository:

- https://github.com/nubomedia/kms-chroma

5.2 PointerDetector

5.2.1 Rationale

This module allows to the user select or throw actions putting a preconfigured object into different regions.

5.2.2 Objectives

In order to develop this module, several objectives were necessaries:

- Develop a GStreamer filter able to follow an object and throw event if the object enters in a preconfigured area.
- Find a way to configure the color of the object to track.
- Define regions in the image and associate images to them.
- Expose the capabilities through the API.

5.2.3 Software architecture

The PointerDetector module has been implemented based on the GstVideoFilter of GStreamer. The object tracking has been done using the OpenCV library. If the images associated to the regions are urls, the module uses soup to download the images.

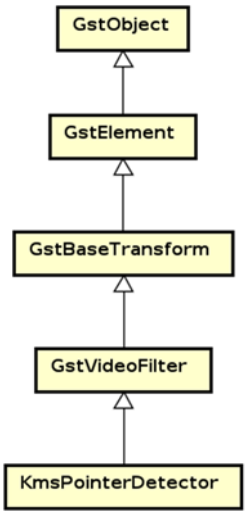

Figure 50: KmsPointerDetector internal structure

5.2.4 Capabilities

Filter capabilities:

- Set the tracking object color. The filter captures the color into a region configured in the module constructor when receives a signal from the client (trackColorFromCalibrationRegion).
- Set the detection regions. It is possible to set regions in the module constructor or in runtime using the method addWindow.
- Remove all windows with clearWindows or only one window with removeWindow.
- Throw a windowIn event when the object to track enters into one of the regions.
- Throw a windowOut event when the object to track leaves one of the regions.

5.2.5 Information for developers

The pointer detector module has been released under the terms and conditions of the LGPL v2.1 license. The source code can be obtained in the following repository:

- https://github.com/nubomedia/kms-pointerdetector

5.3 Augmented Reality media element prototypes

5.3.1 Rationale

NUBOMEDIA task 4.5 for Augmented Reality (AR) cloud elements concentrates on developing and integrating augmented reality capabilities to the NUBOMEDIA platform server (Kurento Media Server i.e. KMS) and thus extent the functionalities of the NUBOMEDIA platform. Augmented reality module(s) support marker based and planar, i.e. 2D image, based augmented reality. The modules are called accordingly ArMarkerTrackerFilter and ArPlanarTrackerFilter. In these modules 2D/3D content can be overlaid on the selected marker or image. The development of the modules is based on ALVAR Desktop which has been developed by the VTT Technical Research Centre of Finland. Rendering of 2D content utilises OpenCV, which is mainly aimed at real-time computer vision and free for use under the open-source BSD license. Rendering of 3D content utilises Irrlicht, which is a free open source 3D engine based on the zlib/libpng license. Also, a filter for visualising values of continuous data has been developed. This module is able to visualise data coming through the Kurento MultiSensory data channel.

5.3.2 Objectives

5.3.2.1 Objectives

In NUBOMEDIA three custom modules for augmented reality (AR) with the API documentation has been developed to support marker based, planar based and multi-domain augmented reality i.e. ArMarkerTrackerFilter, ArPlanarTrackerFilter and MultisensorydataFilterModule.

The main objective for augmented reality task is to provide modules to enable augmentation of the 2D/3D content on video stream. Marker and planar detection, i.e. ArMarkerTrackerFilter and ArPlanarTrackerFilter, called for a suitable AR library and implementation of 2D and 3D content rendering. Also, a way to describe and map targets and content were defined.

MultisensorydataFilterModule needed a way to describe a flexible interface so that various types of AR data can be passed and rendered. New visualisations not yet known can be created thus it is possible to extend the functionality of MultisensorydataFilterModule. An example for drawing a graph is available.

5.3.2.2 ArMarkerTrackerFilter and ArPlanarTrackerFilter

ArMarkerTrackerFilter (Free and open-source software i.e. FOSS) and ArPlanarTrackerFilter (non-FOSS) are augmented reality custom modules for Kurento Media Server. ArMarkerTrackerFilter is based on the ALVAR open source library and the ArPlanarTrackeFilter is based on commercial part of ALVAR and is available for the NUBOMEDIA consortium during the project.

General information of ALVAR augmented reality library can be found from http://virtual.vtt.fi/virtual/proj2/multimedia/alvar/index.html.
ArMarkerTrackerFilter is able to detect ALVAR markers, example in **Error! Reference source not found.**, so that the marker position information can be utilized for rendering. The module detects an ALVAR marker from the video image and based on the detected marker and its pose 2D/3D content can be overlaid (rendered) into the video. Figure 52 shows an example where 2D image and text is rendered over the marker in T-Shirt.
Figure 53 contains an example of 3D rendering on top of the marker. ArMarkerTrackerFilter module is released as open source and can be found from https://github.com/nubomedia/armodule .

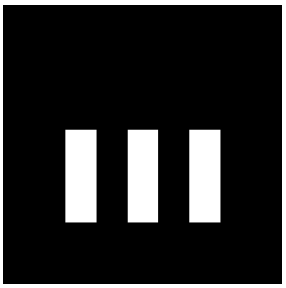

Figure 51. ALVAR Marker with id 251

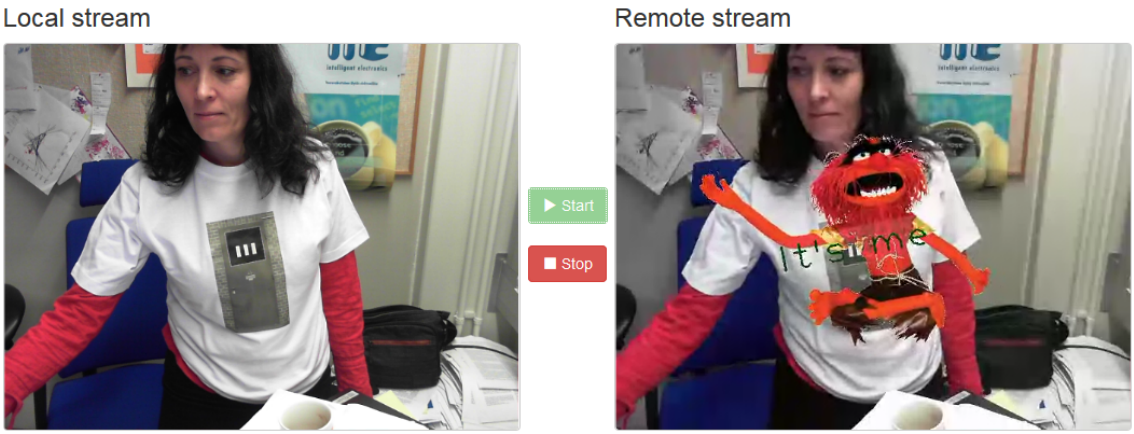

Figure 52. ar-markerdetector example

Example how ar-markerdetector can overlay in video stream text/image on top of the mark

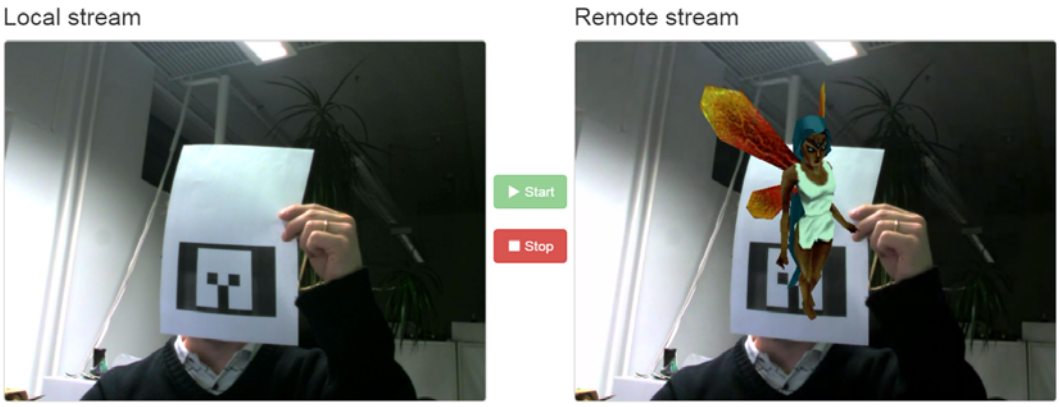

Figure 53. Example how ar-markerdetector can overlay 3D images on top of the marker

The ArPlanarTrackerFilter can recognize and track planars. . The planar is given to the filter as input and in principle it can be any image although the image must be contain features suitable for ALVAR that the ArPlanarTrackerFilter can track. Also in this case 2D/3D images can be rendered on top of the planar. Example of ArPlanarTracker Filter is in Figure 54. ArPlanarTrackerFilter is not released as open source but it is available for the consortium during the project.

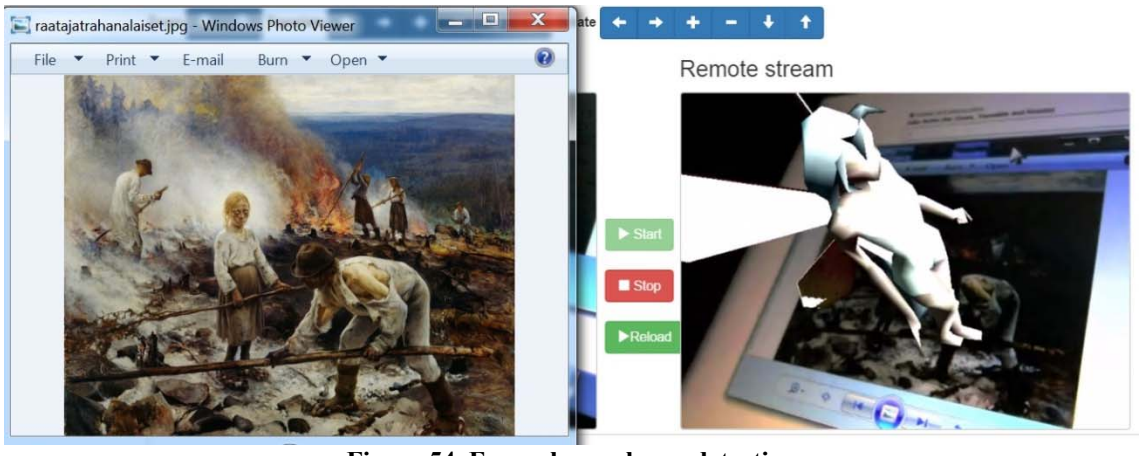

Figure 54. Exmaple on planar detection

On the left is the planar to be detected, on the right a 3d model is overlayed on the found planar image.

5.3.2.3 MultisensorydataFilter

MultisensorydataFilter (Free and open-source software i.e. FOSS) is an augmented reality custom module for Kurento Media Server. Rendering of 2D content utilises OpenCV.

MultisensorydataFilter implements multi-domain AR filter providing 2D graphics visualisation service through data pads and also with data channels. As a result, MultisensorydataFilter interface enables receiving of different kinds of content and based on the interpretation of the data, i.e. the format of the data, the interface is quite flexible and supports future extensions of adding more visualization capabilities to the filter.
Because of the flexible interface more visualization methods can be created in the future, while maintaining the compatibility with older applications. Figure 55 contains an example of 2D rendering a graph based on the values received from the web browser.

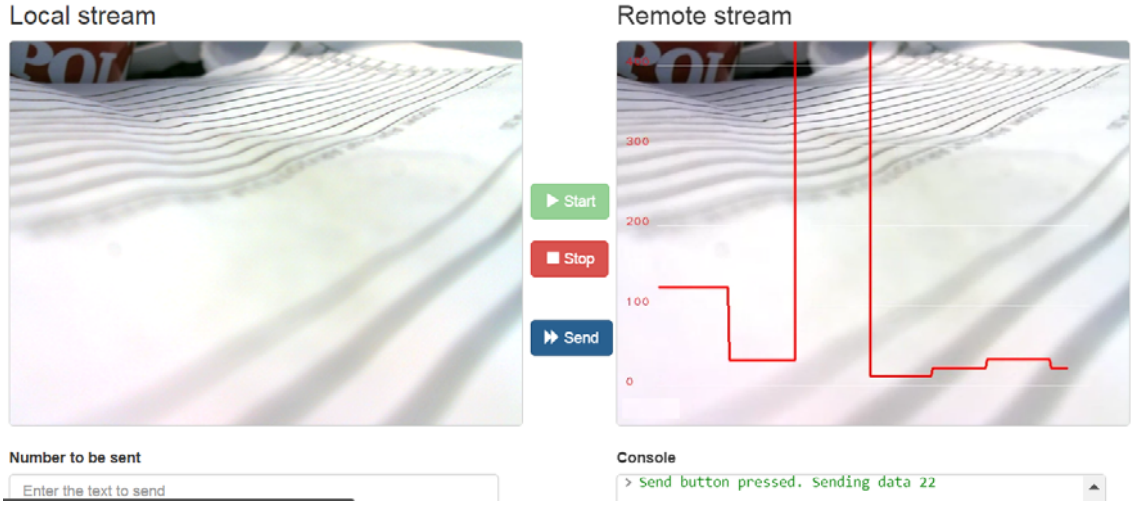

Figure 55. Example how MultisensorydataFilter can overlay sensor data on video stream

5.3.3 Software Architecture

5.3.3.1 Arfilters

ArMarkerTrackerFilter and ArPlanarTrackerFilter are developed as Kurento Media Server modules. Figure 56 shows the class diagram for the key parts. The modules are based on OpenCV and the basic code for the implementation is generated automatically by Kurento based on the module specific filter interface. Refer the Kurento documentation (http://kurento.com/) for a more complete view of the Kurento architecture. We have the basic augmented reality processing in a separate `ArProcess` class so that the core implementation is safe, even if interfaces change and we need to re-generate the code. Basically the structure is similar for both ArMarkerTrackerFilter and ArPlanarTrackerFilter but they differ if ArProcess utilizes alvar::MarkerDetector or alvar::PlanarDetector to detect Alvar markers or planar images. alvar::MarkerDetector utilises ALVAR Desktop (See Figure 57), which again uses the OpenCV computer vision library and alvar::PlanarDetector utilises a proprietary product. RenderingEngine utilises currently Irrlicht, and in detail, the filtering steps for the process is following:

1. An input video frame as OpenCV image is passed to the RenderingEngine
2. The image is rendered to the Irrlicht render target texture format
3. A 3D model is drawn on top of the render target texture
4. 2D snapshot of the 3D scene view is generated

5. Snapshot image is the output frame of the ArMarkerTrackerFilter

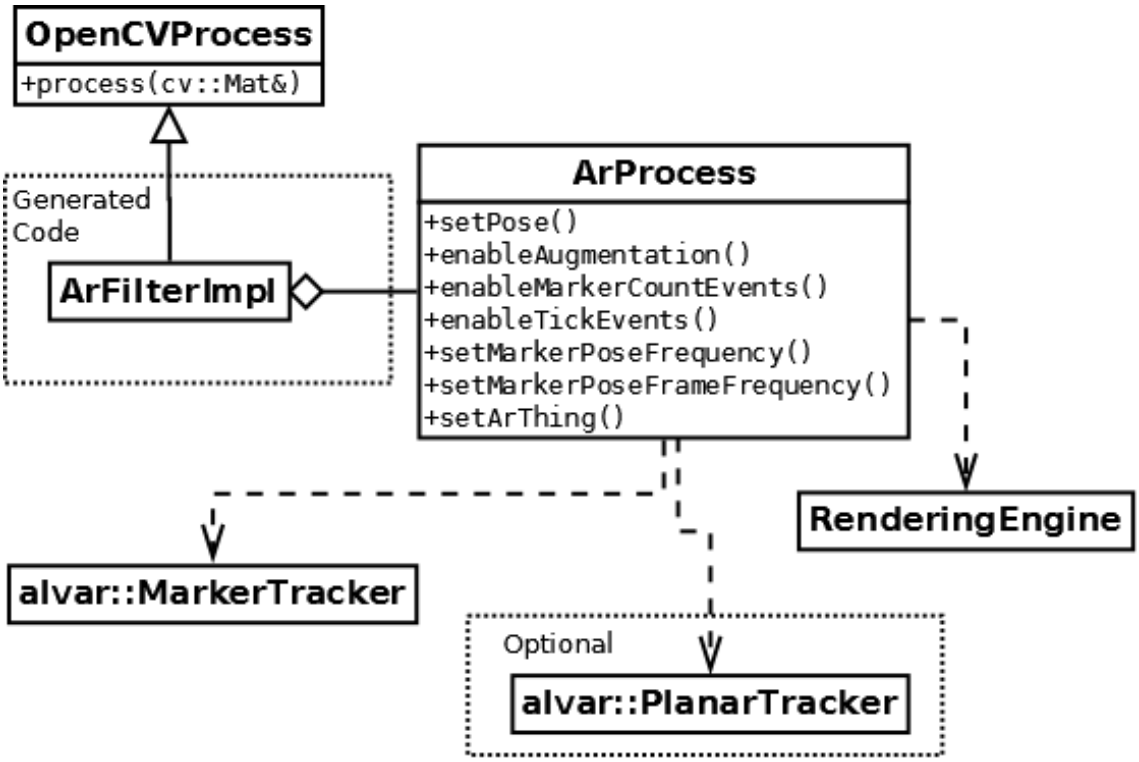

Figure 56. The relations of key components in ar-markerdetector.

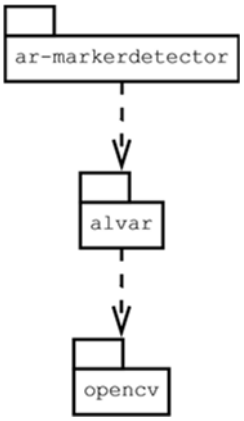

Figure 57. ar-markerdetector module dependencies

5.3.3.2 Multisensorydatafilter

MultisensorydataFilterModule (msdata in Nubomedia git repository, https://github.com/nubomedia/msdata) enables receiving of arbitrary data through data pads from other modules also originated from DataChannels for visualisation purposes. Because the filter needs to support data-pads, the filter is based on Kurento GStreamer module. MultisensorydataFilterModule receives all video, data and other data through its data pads. Also, when utilized with DataChannels all the data that the WebRtcEndpoints has received from DataChannels ie the out pads of the WebRtcEndpoints is connected to the in pads of MultisensorydataFilterModule. The development of the module is based on Kurento GStreamer module. Refer the Kurento documentation (http://www.kurento.org/) for a more complete view of the Kurento architecture.

Figure 58 shows the class diagram for the key parts of MultisensorydataFilterModule. The data pad that is utilized is data ie the other ones were video and audio. The process for sending data from other modules is connecting their output data pads to the input data pad of MultisensorydataFilterModule. MultisensorydataFilterModule can now pass the data for the selected RenderingEngine.

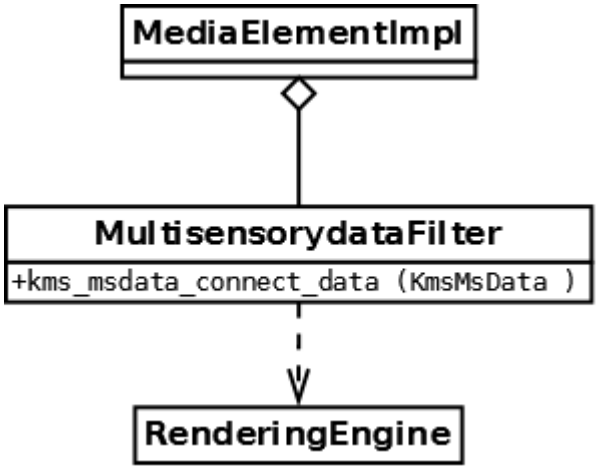

Figure 58. Relations of Key Components

5.3.4 Capabilities

5.3.4.1 ArMarkerTrackerFilter and ArPlanarTrackerFilter

In this section the API for both ArMarkerTrackerFilter and ArPlanarTrackerFilter filters in Nubomedia is described. The private repository for ARPlanarTrackerFilter utilizes the FOSS implementation ArMarkerTrackerFilter so that FOSS repository is cloned and planar detection implementation is injected into FOSS implementation. ArMarkerTrackerFilter contains no-operation planar tracking functionality and ArPlanarTrackerFilter replaces this with functional Alvar planar detection implementation.

The API for both FOSS and proprietary implementation are equal. But, the planar detection is enabled with “detect_planar” parameter in setArThing methdod so that the planar path and name is given as input. See Table 1 and Table 2 for the methods and events.

Following API description for ARfilters can be found from the JSON description in https://github.com/nubomedia/armodule/blob/master/ar-markerdetector/src/server/interface/armarkerdetector.ArMarkerdetector.kmd.json and example of using them in https://github.com/nubomedia-vtt/armoduledemos/blob/master/DevelopersGuide.md.

Table 1. Methdos for ArMarkerTracker and ArPlanarTracker

Method	Description

enableAugmentation	Sets the enabled state of the 2D and 3D rendering
enableMarkerCountEvents	Sets the enabled state for notifying the listener when the number of markers changes
enableTickEvents	Sets the enabled state for notifying the listener when number of ticks has elapsed for monitoring

setArThing	Passes set of augmentables ie info about markers/planars that should be tracked and 2D/3D content/models to be rendered. Following parameters can be given as input:
	
	`"id":0 = Alvar Marker 0 ie Zero Marker`
	`"type":"3D" = render as 3D model (2D for flat models)`
	`"model":"/opt/faerie.md2" = 3D model that to be rendered`
	`"texture":"/opt/faerie2.bmp" = texture for the 3D model in md2 format`
	`"scale":0.09} = scaling of the 3D model`
	
	When using ArPlanarTracker `"detect_planar"` key is used that describes the planar that should be tracked.

setMarkerPoseFrequency	Sets the enabled state for marker events and how often these events should be generated based on time
setMarkerPoseFrameFrequency	Sets the enabled state for marker events and how often these events should be generated based on number of processed frames
setPose	Sets operation i.e. position/rotation/scale of 3D model

Table 2. Events for ArMarkerTracker and ArPlanarTracker

Event	Properties	
MarkerCount		An event that is sent when the number of visible markers is changed.
	markerID	ID number of detected marker
	markerCount	Number of visible markers with the specified id
	markerCountDiff	Delta of markercount compared to previous event
	sequenceNumber	
	countTimestamp	
MarkerPose		Matrices for marker pose
	sequenceNumber	
	poseTimestamp	
	width	Integer to tell the width of the input video e.g. for initializing the AR data

		structures
	height	Integer to tell the height of the input video e.g. for initializing the AR data structures
	matrixProjection	
	markerPose	

5.3.4.2 MultisensorydataFilter

In this section the API for MultisensorydataFilter filter in Nubomedia is described.

Table 3. Methdos for ArMarkerTracker and ArPlanarTracker

Method	Description
kms_msdata_connect_data	Deliver arbitrary data through data pad

Currently data pad is utilized as the interface to deliver data to MultisensorydataFilterModule as shown in Table 3. Thus, currently the functionality is driven through interpretation of the given data. Explicit functionality can be added as necessary. The current implementation interprets the given data as a series of integers that are visualized as a graph as shown in Figure 55.

5.3.4.3 Evaluation and validation

5.3.4.3.1 Evaluation of the module performance

We profiled ArMarkerTrackerFilter and ArPlanarTrackerFilter by measuring how much time is elapsed between some key points inside of the module in order to understand the performance of different functionality inside the AR module. The functionality includes the detection of AR marker or planar surface as well as rendering the 3D content on top of the image. As a result, we were able to find out how much each key functionality of the module consumes the time resources and finally to put effort where we could benefit most on optimising the implementation. Figure 59 shows a flow diagram of the AR module and explains the meaning of different measurement points listed as:

- entering and leaving the module (ARFILTER(ONLY))
- Combined marker and planer detection (ALLDETECTED)
- 3D Rendering (ALL AUGMENTED)
- Time between the consecutive calls of the AR module (KMS (WITHOUT AR FILTER)

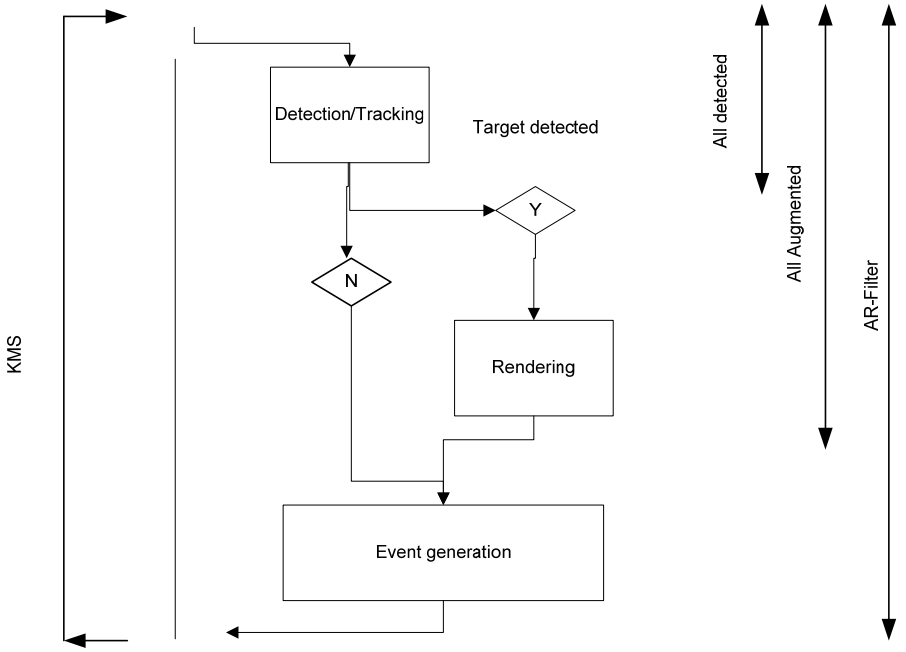

Figure 59. Flow diagram of the AR module and time measurement points

Figure 60 and Figure 61 presents examples of the measurement results. The major grid lines are 20ms each and the processing time of the AR module should remain under 100ms for obtaining lag free processing in frame rate of 10fps that was used in the measurements. Figure 5.2 presents results obtained in non-GPU accelerated server hardware (Core i7, 2.7 GHz). Figure 5.3 presents results in same server hardware (Core i7, 2,7 GHz) with and additional GPU accelerated 3D rending hardware.

Both test cases contain measurement in two different processing flows. First the marker or planar surface is not visible in the scene, thus most of the time is consumed by marker/planar detector while 3D rendering does not contribute at all, since 3D content is not rendered until the AR tracking target has been found.

After some time (time 617 in Fig. 5.2 and 1114 in fig. 5.3) the target is found and 3D rendering starts, thus increasing the overall processing requirements of the module.

Figure 5.2 shows that rendering takes most of the time in non-3D accelerated system (all augmented) even though the models are pretty simple, a teapot and a faerie. The system can hardly keep up with the real-time performance as AR-module is consuming 80ms out of the 100ms time window available for lag free performance.

A large performance improvement was gained with using accelerated 3D hardware as Figure 5.3 shows. While rendering, the AR module is consuming only 20ms for tracking the target and rendering the content. As a result, GPU should be utilized for rendering instead of CPU for rendering if possible.

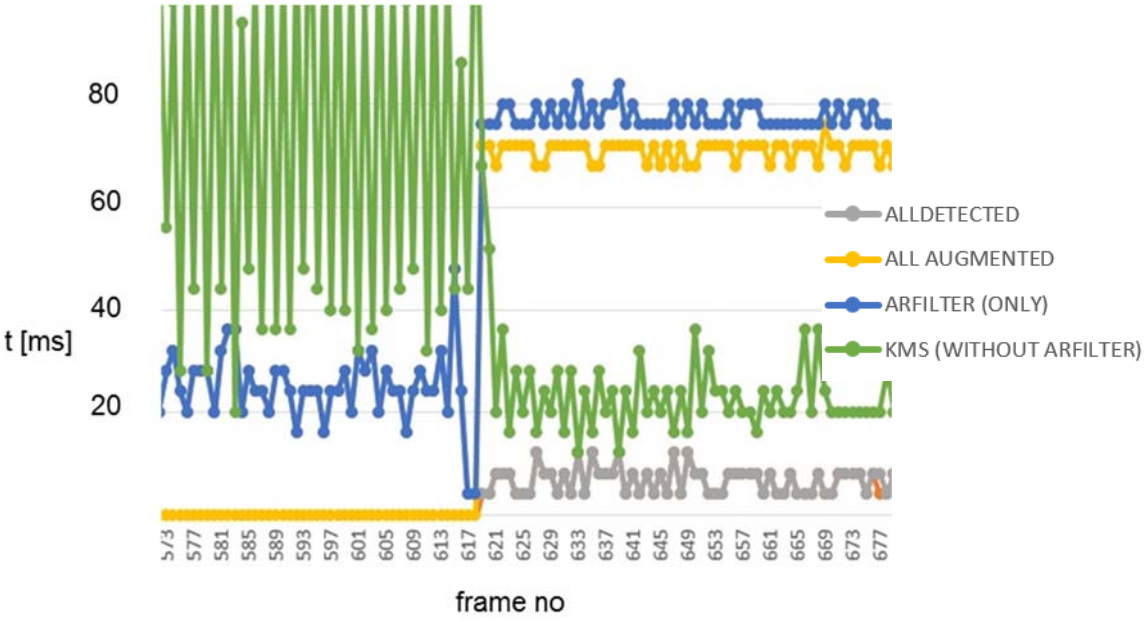

Figure 60 Results obtained with non GPU accelerated server hardware

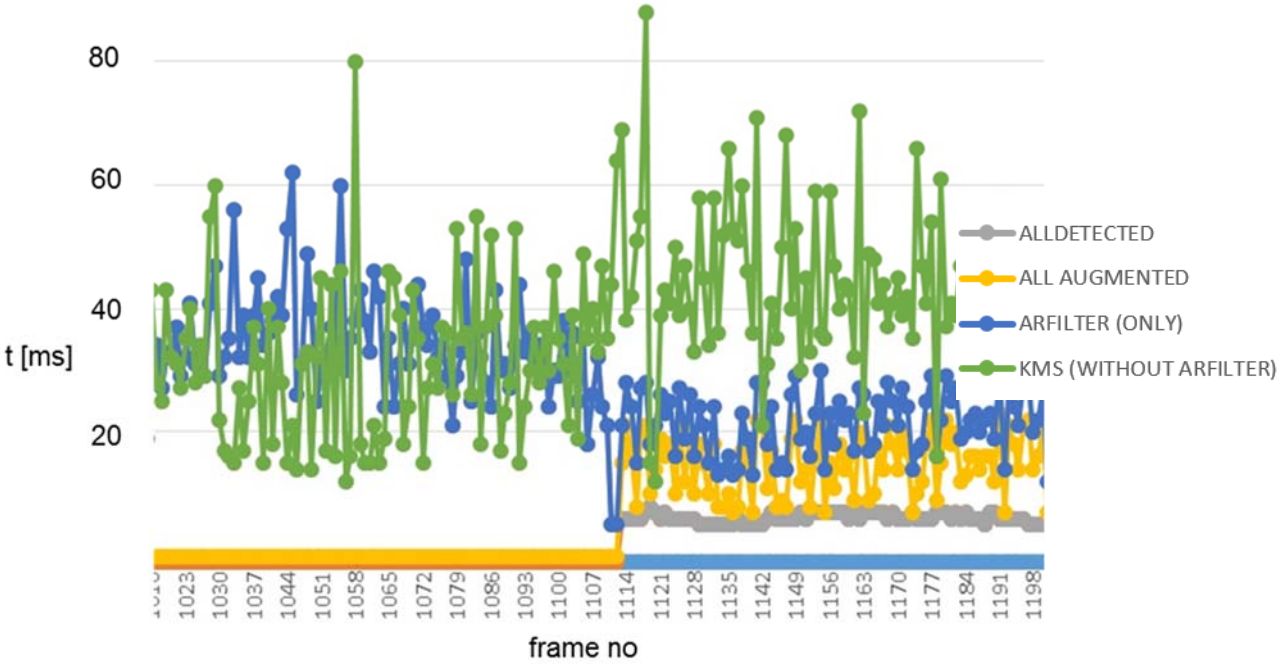

Figure 61 Results obtained with GPU accelerated server hardware

5.3.4.3.2 Evaluation of End to end video latency

The measurements of end to end video latency were performed with the web-RTC statistics enabled in Kurento media server in the same measurement setting as the evaluation in section and using GPU accelerated 3D rending hardware. The results are show in the Figure 62. The Planar detection is naturally more time consuming and generates higher latency peaks from time to time. Also, the effect of rendering is more clearly seen from the marker detection module as the marker detection and tracking does not consume CPU as much as planar detection and tracking.

Apart from the one high peak at the planar module the performance stays below 100 ms to support real time requirement. The peak at the beginning is caused by the first rendering sequence relating to initialization of the Irrlicht.

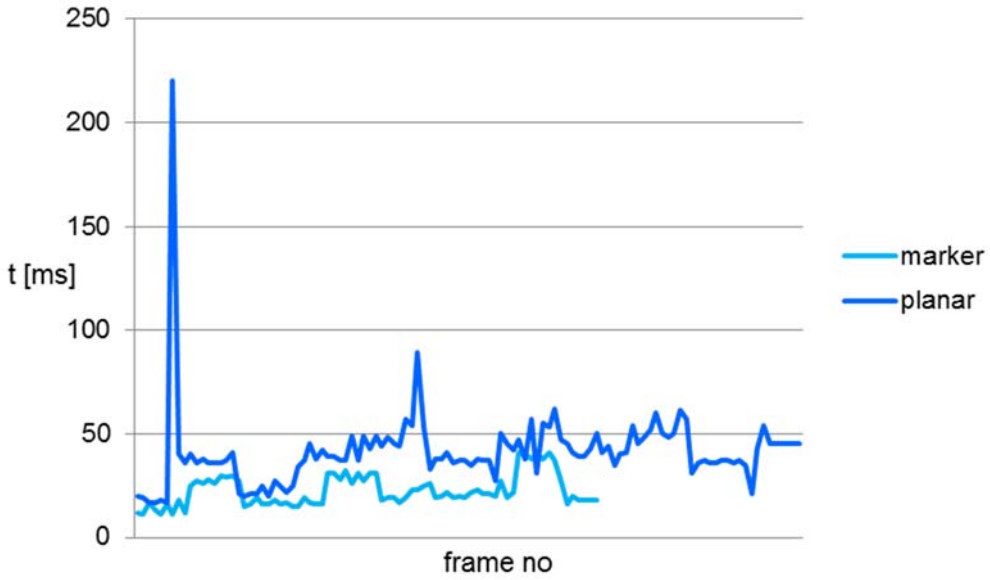

Figure 62 E2E video latency for AR modules
Light blue line for marker based and dark blue line for planar based.

The average E2E time for marker based AR and planar based AR are in Table 2.
The measurement show that the ARPlanarTrackerFilter consumes more time, but even with the high load peak in the beginning of the planar graph the average of both filters are reasonable considering real time requirements.

Table 2 Average E2E times for AR modules

Method	Average E2E time [ms]
ARMarkerTrackerFilter	22, 57
ARPlanarTrackerFilter	41,33

5.3.5 Information for developers

The ArMarkerTrackerFilter has been released under the terms and conditions of the Apache License, Version 2.0.
The source code can be obtained in the following repository:
https://github.com/nubomedia/armodule.
The documentation, installation guide and developers guide for ArMarkerTrackerFilter are available at http://nubomedia-vtt-ar.readthedocs.io/en/latest/.
An example code for using the ArMarkerTrackerFilter module can be found from https://github.com/nubomedia/arfilterdemopaas and corresponding documentation http://nubomedia-vtt-ar-demo.readthedocs.io/en/latest/.

ArPlanarTrackerFilter is not released as open source but it is available for the consortium during the project. Information how to get access on it is found from here https://docs.google.com/document/d/1PoRZQG5YEjEHSIhvYXq8Gmd9fajZdU75Z1APDLHYUKI/edit?pref=2&pli=1#.

The MultisensorydataFilterModule has been released under the terms and conditions of the Apache License, Version 2.0. The source code can be obtained in the following repository:

https://github.com/nubomedia/msdata. The documentation, installation guide and developers guide for MultisensorydataFilterModule are available at http://nubomedia-vtt-msdata.readthedocs.io/en/latest/.
An example code for using the MultisensorydataFilterModule can be found from https://github.com/nubomedia/msdatademopaasgraph .

5.3.5.1 Known Issues in ArMarkerTrackerFilter and ARPlanarTrackerFilter

When ARMarkerTrackerFilter is running in a Linux machine where X-windows graphical user interface is used, the user interface may freeze which can cause unfavourable situation. The freezing of X-window do not prevent of using the AR module. Therefore, one should have separate server for running Kurento server including ArMarkerTrackerFilter/ArPlanarTrackerFilter where the connection is taken through e.g. SSH client. This is mainly an issue in the development phase as in real deployment the Kurento server is always running in the different hardware setup as the connecting clients. The developers has investigated how to fix the freezing of X system and as a solution the following can be utilised.

```
sudo apt-get install xserver-xorg-video-dummy
wget xpra.org/xorg.conf
Xorg -noreset +extension GLX +extension RANDR +extension RENDER -logfile ./0.log -
config ./xorg.conf :0
```

If you have problems with port, just try another eg:

```
Xorg -noreset +extension GLX +extension RANDR +extension RENDER -logfile ./10.log -
config ./xorg.conf :10
```

5.4 Video Content Analysis modules

The idea of these modules is to create different VCA algorithms as a proof of concept to be integrated in the NUBOMEDIA platform. These algorithms will be selected based on the list obtained in the WP2 which shows the services and functionalities that solve the customer problems and needs. The list with the different services is given below.

Requirement	Priority	Use Case	Owner
Nose, mouth and ears detection.	High	UC-ZED	VTOOLS
Motion detection	Low	UC-VT-1, UC-TI	VTOOLS
Object tracking	Medium	UC-VT-1/2 UC-LU, UC-NT, UC- TI	VTOOLS
Intrusion detection	Low	UC-VT-1/2, UC-TI	VTOOLS
3D – extracting depth info	Low	UC-VT-1/2,	VTOOLS
Statistics for retail sector	Low	UC-VT-2	VTOOLS
Automatic lip reading	Low	UC-LU	VTOOLS
Lip extraction and color characterization	Low	UC-NT	VTOOLS

Face extraction and color characterization	Low	UC-NT	VTOOLS
Breath frequency characterization from chest movements	Low	UC-NT	VTOOLS
Walking movement characterization	Low	UC-NT	VTOOLS
Skin rash characterization	Low	UC-NT	VTOOLS
Spasms characterization from body movements	Low	UC-NT	VTOOLS
Heartbeat characterization from skin color variations	Low	UC-NT	VTOOLS
The system to be able to trigger events, based on video's feature	Medium	UC-TI	VTOOLS
Face detection and counting faces	High	UC-VT-1/2, UC-ZED, UC-TI	VTOOLS
Identify the place where the objects have been hidden	High	UC-ZED-2	VTOOLS
Detect and track other facial features	Low	UC-ZED	VTOOLS

Table 3: VCA Requirements from the D2.2.1

Due to the high number of requirements, 18 requirements for Video Content Analysis, and the time it takes to develop these algorithms in a robust manner, we are forced to choose some of them based on the following:

- The Description of Work (DoW).
- Strategic decision, by which we try to develop algorithms that can serve us for new business opportunities.
- Based on the frequency that these algorithms are used.
- The difficulty of the algorithms. Since we need to cover several algorithms to be used by the different partners on task 6.2, we cannot invest an unreasonable time for a specific algorithm, because we need to cover a reasonable number of filters so that all the partners can use in their demonstrator a VCA filter if they wish.
- The more mature computer vision algorithms currently used in real applications, which are all of them well-known

Then, we are going to explain what are the different algorithms developed in every release.

Release 2 (R2)

This second release covers from M5 (06/2014) to M8 (09/2014). During this release, we have realized the following work.

- VT's team has begun to learn Kurento platform which will be used on NUBOMEDIA to generate the different media elements and pipelines. In this case, we have learnt how to develop the VCA plugins and how integrate them in the Kurento platform.

- The requirement motion and face detection have been developed, you can read more about this algorithm in this deliverable.

Release 3 (R3)

The third release covers from M9(10/2014) to M12(01/2015). During this release we have realized the following work.

- VT's team has carried out the necessary changes in the algorithms already developed (motion and face detector) to adapt them to the new changes of the Kurento platform.
- The nose, mouth and ear requirement which can be split into three different algorithms (Nose, mouth and ear detector) has been developed, you can read more about this algorithm in this deliverable.
- All the algorithms developed up to now have been integrated on the first version of NUBOMEDIA's cloud platform.
- Finally, some examples of demonstrator which combine different VCA filters have been developed.

Release 4 (R4)

The fourth release covers from M13(02/2014) to M16(05/2015). During this release we have realized the following work

- The Tracker algorithm, which is a widely used in industrial applications, and have been demanded by some of the partners of this consortium, as you can see on the previous section.
- The Intrusion detection through which we can detect outdoors perimeter intrusions.
- Visual Tools has also developed two new applications in order to test the algorithms developed on this release.
- Both algorithms have been integrated over the NUBOMEDIA's cloud platform.
- Finally, some minor changes have been carried out in the entire demonstrator developed due to changes in the NUBOMEDIA platform.

Release 5 (R5)

The fifth release covers from M17(06/2014) to M16(09/2015). During this release we have realized the following work:

- The requirement eye detector have been developed, you can read more about this algorithm in this deliverable.
- In addition, we have developed a new demo in order to show the functionality of the eye detector algorithm.
- VT's team has carried out the necessary changes in all the algorithms already developed to adapt them to the new changes of the Kurento platform, migration from version 5 to version 6.
- All the new versions of the algorithms have been developed on the new version of the NUBOMEDIA platform.

Release 6 (R6)

The sixth release covers from M17(10/2014) to M16(01/2016). During this release we have realized the following work:

- The video end to end latency, showing the time it takes an image from entering the pipeline until it exits, has been introduced in all the demonstrators.
- We have worked on improving all the algorithms related to detection of some facial segment.

5.4.1 VCA modules architecture

In this section, we are going to see a high-level architecture explaining the general overview of the architecture to develop the different VCA algorithms. The following figure depicts the base architecture to develop the different VCA algorithms over the NUBOMEDIA infrastructure.

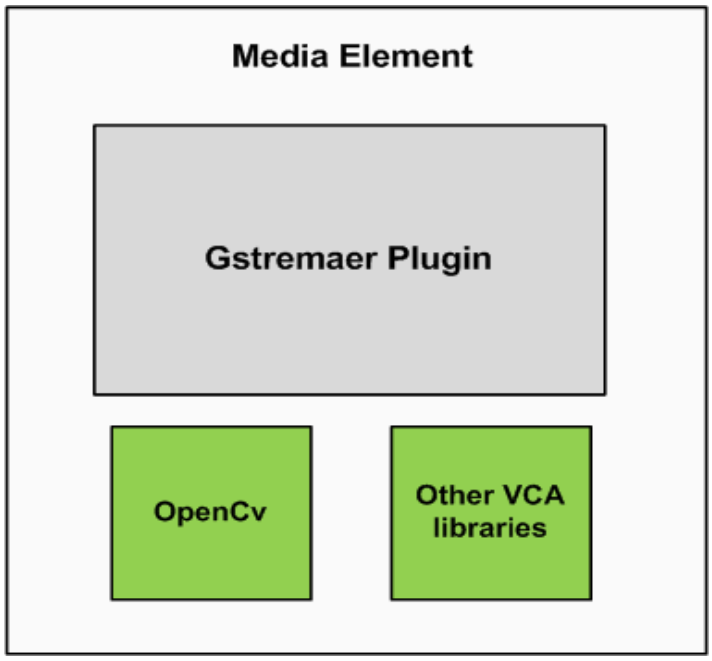

Figure 63: General VCA architecture

The different algorithms are integrated in NUBOMEDIA through the Media Elements (ME). These media elements are monolithic and abstractions elements which send and receive media streams and data through different protocols. The goal of these elements is to carry out specific processing of the media stream or the data. Specifically, for this task the media elements are focused on applying VCA algorithms in order to extract valuable semantic information. If you are interested on knowing more about Media Elements, please see the D2.4 “NuboMedia Architecture”.

After seeing about which elements of the NUBOMEDIA architecture the VCA algorithms are integrated, we will see the modules used to integrate these algorithms and process the video and data. The Media Elements offer the possibility to develop a specific algorithm through a GStreamer plugin. These plugins offer to the developers the video frame by frame in a specific function. Therefore, the user can perform the processing of the frame in this function. With the aim to process every frame the developers can use different libraries. On this project a high percentage of the algorithms are going to be developed using the OpenCV library. On the other hand, an important percentage of the algorithms are going to be developed using other libraries.

Finally, once the algorithm has been developed and installed, the developers can use the filter through the Kurento platform using the Java or JavaScript API.

5.4.2 VCA Modules

Then, the developed modules are described

5.4.2.1 Nose, Mouth, ear, eye and face detection

First of all it is important to highlight that all this algorithms are included in the same section, since their basic technology is based on the same algorithm.

5.4.2.1.1 Rationale and objectives

These algorithms have been selected to be included on the NUBOMEDIA platform for the following reasons:

- They are an algorithm which is widely used in industrial applications, and we believe that it could be of great use for the NUBOMEDIA partners and possible future users of the platform.
- Two of the partners in the project are interested in it.
- Furthermore, this requirement has a high priority for them.
- There is lot information about the technology in nose, mouth, ear and face detection which allows us to develop them in reasonable time.

We are interested in developing them for a possible industrial exploitation.

The main goal of these algorithms is to detect different part of the face to be used for the demonstrator of WP6. For example, some partners are interested on detecting the different parts of the face to use the Augmented Reality filters to overlay different things over them. For example, we can use the eye detector to superimpose eyeglasses. Another example, it is the option to use the face detector to make possible the smart search in the video surveillance demonstrator.

5.4.2.1.2 Software Architecture

For these algorithms, we are using the **OpenCV library.** All of them are based on the same algorithm. Specially, we take advantage of the **Haar featured-based Cascade classifier** included in this library.

This Object Detection method is an effective algorithm proposed by Paul Viola and Michael Jones in their paper, “Rapid Object Detection using a Boosted Cascade of Simple Features” [VIOLACVPR]. The three main characteristics of this detection algorithm are:

1. A new image representation call ***integral image*** through which the image can be computed from an image using a few operations per pixels.
2. Build a classifier using a small number of important features using AdaBoost (adaptive boosting, a machine learning meta-algorithm). This **classifier** deletes the majority of the features and it is focused on a small set of critical features.
3. Combine successively more complex classifiers in a cascade (known as **cascade of classifiers**) structure which increases a lot the speed of the detector.

The features, which we have named on the previous list, are small region of the image with a single value obtained by subtracting sum of pixels under white rectangle from sum of pixels under black rectangles. You can see an example in the following image, in this image we can find the three kinds of features that this algorithm uses. The first feature is two rectangle feature showed in (a), the second one is a three-rectangle feature showed in (b) and the third one is a four-rectangle feature showed in (c).

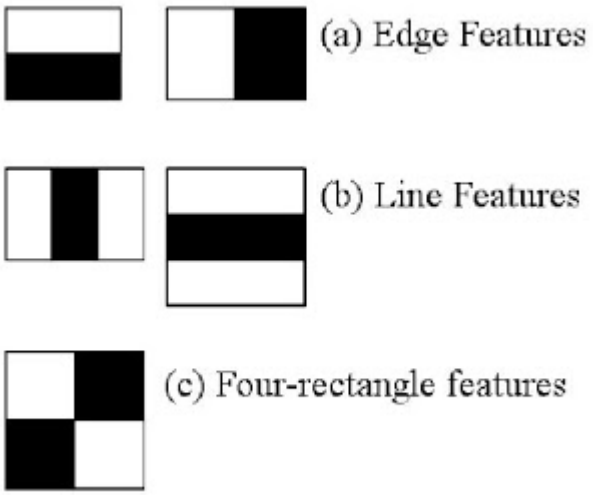

Figure 64: Features of the Haar cascade classifier

Integral image

All possible sizes and locations of each part of the image are used to calculate plenty of features. (Just imagine how much computation it needs? Even a 24x24 window results over 160000 features). For each feature calculation, we need to find sum of pixels under white and black rectangles. To solve this, they introduced the integral image, which simplifies the calculation of sum of pixels, involving just four pixels. This makes the calculations super-fast. In more detail, the integral image at location *x,y*, contains the sum of the pixels above and to the left of *x,y*, inclusive.

Where ii(x,y) is the integral image and i(x,y) is the original image. For example, the sum of the pixels within the rectangle D (see the following image) computed with four array reference. The value of the integral image at location 1 is the sum of the pixels in rectangle A. The value of location 2 is A + B, at location 3 is A + C, and at location 4 is A + B + C + D. Therefore, using the integral image any rectangle can be computed in four array references.

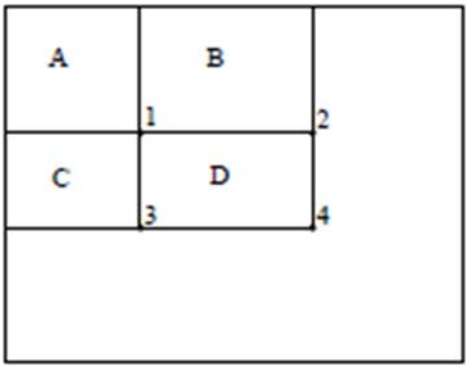

Figure 65: Face detection integral image

Learning classification functions

In this stage, the algorithm is trained from a lot of positive (images of faces) and negative images (images without faces). The algorithm looks for the best rectangle feature which separates the positive and negative images.

For example, in the case of face detection, the best features are:

1. The first feature measures the difference in intensity between the region of the eyes and a region across the upper cheeks.
2. The second feature compares the intensities in the eye regions to the intensity across the bridge of the nose.

In the following figure, we can see a graphical representation of these two areas.

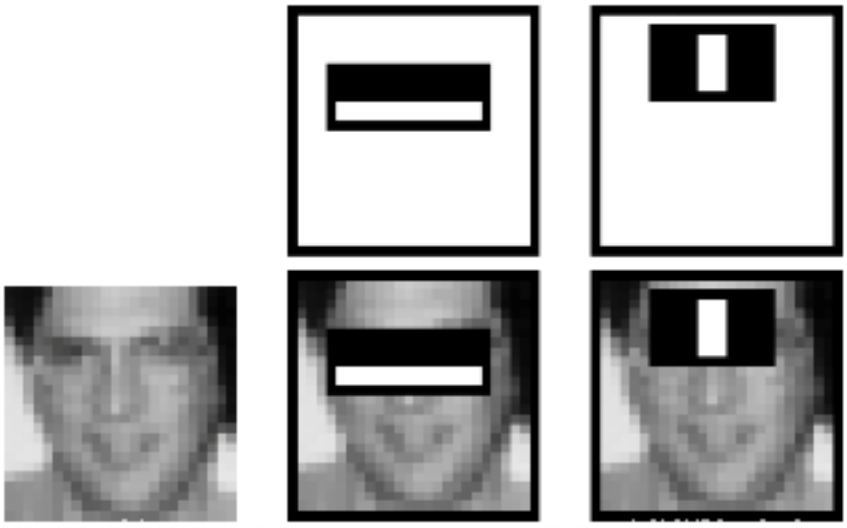

Figure 66: Best features for Face recognition

Cascade of Classifiers

The following figure illustrates the cascade of classifiers. The first classifier is applied to every sub-window. This first classifier eliminates a large number of negative examples with very little processing. Those sub-windows which are not rejected by this classifier are processed by a sequence of classifiers, each slightly more complex than the last. If any classifier rejects the sub-window, no further processing is performed to this sub-window.

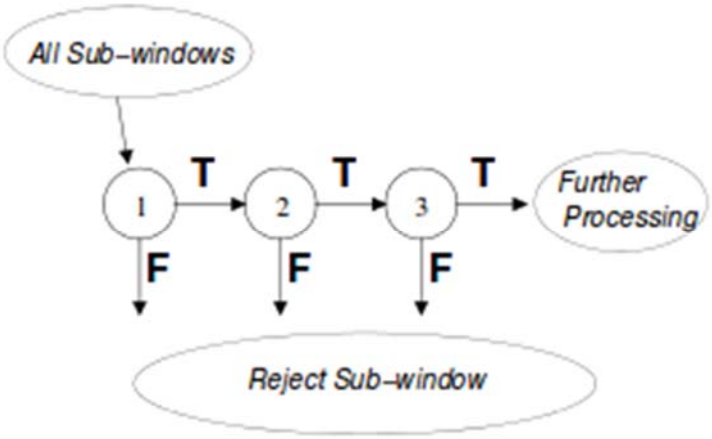

Figure 67: Cascade of classifiers

Once we have seen the main components for these algorithms, we will see the different API for each algorithm.

Api for Face detector

The developers can use the following filter's API for its setup:

Function	Description
void showFaces (int);	To show or hide the bounding boxes of the detected faces within the image. Parameter's value: - 0 (default), the bounding boxes will not be shown. - > 0, the bounding boxes will be drawn in the frame.
void detectByEvent (int);	To indicate to the algorithm if it must process all the images or only when it receives a specific event such as motion detection. Parameter's value: - 0 (default) , process all the frames; - 1, process a frame when it receives a specific event
void sendMetaData (int);	To send the bounding boxes of the faces detected to another ME as a metadata. Parameter's value: - 0 (default), metadata are not sent. - 1, metadata are sent.
void withToProcess(int)	This will be the width of the image that the algorithm is going to process to detect the faces. A higher value produces better results but the processing time will be higher. A good parameter value will be 160.
void processXevery4Frames(int)	Through this method we indicate to the algorithm to process x frames every 4 frames. The x value can be: - 0: process 0 images and discard 4. It is like working at 0 fps. - 1: process 1 image and discard 3 (8 fps). - 2: process 2 images and discard 2 (12 fps).

	- 3: process 3 images and discard 1 (18 fps). - 4: process 4 images and discard 4 (24 fps)

Table 4: Face detector API

Api for Nose detector

Developers can use the filter's API for its setup. Then, the API is shown:

Function	Description
void viewNoses (int);	To show or hide the bounding boxes of the detected noses within the image. Parameter's value: - 0 (default), the bounding boxes will not be shown. - > 0, the bounding boxes will be drawn in the frame.
void detectByEvent (int);	To indicate to the algorithm if it must process all the images or only when it receives a specific event such as face detection. Parameter's value: - 0 (default) , process all the frames; - 1, process a frame when it receives a specific event
void sendMetaData (int);	To send the bounding boxes of the noses detected to another ME as a metadata. Parameter's value: - 0 (default), metadata are not sent. - 1, metadata are sent.
void multiScaleFactor(int);	To generate the cascade of classifier, the algorithm creates a pyramid iteration scale, in every iteration the image is reduced. Through this method, the user can specify how much the image size is reduced at each image scale. A higher value the processing time will be reduced and the robustness of the algorithm will decrease. A good parameter value will be 20. The value by default is 25.
void withToProcess(int)	This will be the width of the image that the algorithm is going to process to detect the nose. A higher value produces better results but the processing time will be higher. A good parameter value will be 320.
void processXevery4Frames(int)	Through this method, we indicate to the algorithm to process x frames every 4 frames. The x value can be: - 0: process 0 images and discard 4. It is like working at 0 fps. - 1: process 1 image and discard 3 (8 fps). - 2: process 2 images and discard 2 (12 fps). - 3: process 3 images and discard 1 (18 fps). - 4: process 4 images and discard 4 (24 fps)

Table 5: Nose detector API

Api for Mouth detector

Developers can use the filter's API for its setup. Then, the API is shown:

Function	Description
void viewMouths (int);	To show or hide the bounding boxes of the detected mouths within the image. Parameter's value: - 0 (default), the bounding boxes will not be shown. - > 0, the bounding boxes will be drawn in the frame.
void detectByEvent (int);	To indicate to the algorithm if it must process all the images or only when it receives a specific event such as face detection. Parameter's value: - 0 (default) , process all the frames; - 1, process a frame when it receives a specific event
void sendMetaData (int);	To send the bounding boxes of the mouths detected to another ME as a metadata. Parameter's value: - 0 (default), metadata are not sent. - 1, metadata are sent.
void multiScaleFactor(int);	To generate the cascade of classifier, the algorithm creates a pyramid iteration scale, in every iteration the image is reduced. Through this method, the user can specify how much the image size is reduced at each image scale. A higher value the processing time will be reduced and the robustness of the algorithm will decrease. A good parameter value will be 20. The value by default is 25.
void withToProcess(int)	This will be the width of the image that the algorithm is going to process to detect the mouth. A higher value produces better results but the processing time will be higher. A good parameter value will be 320.
void processXevery4Frames(int)	Through this method, we indicate to the algorithm to process x frames every 4 frames. The x value can be: - 0: process 0 images and discard 4. It is like working at 0 fps. - 1: process 1 image and discard 3 (8 fps). - 2: process 2 images and discard 2 (12 fps). - 3: process 3 images and discard 1 (18 fps). - 4: process 4 images and discard 4 (24 fps)

Table 6: Mouth detector API

Api for Ear detector

Developers can use the filter's API for its setup. Then, the API is shown:

Function	Description
void viewEars(int);	To show or hide the bounding boxes of the detected ears within the image. Parameter's value: - 0 (default), the bounding boxes will not be shown.

	- > 0, the bounding boxes will be drawn in the frame.
void detectByEvent (int);	To indicate to the algorithm if it must process all the images or only when it receives a specific event such as face detection. Parameter's value: - 0 (default) , process all the frames; - 1, process a frame when it receives a specific event
void sendMetaData (int);	To send the bounding boxes of the mouths detected to another ME as a metadata. Parameter's value: - 0 (default), metadata are not sent. - 1, metadata are sent.
void multiScaleFactor(int);	To generate the cascade of classifier, the algorithm creates a pyramid iteration scale, in every iteration the image is reduced. Through this method, the user can specify how much the image size is reduced at each image scale. A higher value the processing time will be reduced and the robustness of the algorithm will decrease. A good parameter value will be 20. The value by default is 25.
void withToProcess(int)	This will be the width of the image that the algorithm is going to process to detect ears. A higher value produces better results but the processing time will be higher. A good parameter value will be 320.
void processXevery4Frames(int)	Through this method, we indicate to the algorithm to process x frames every 4 frames. The x value can be: - 0: process 0 image and discard 4. It is like working at 0 fps. - 1: process 1 image and discard 3 (8 fps). - 2: process 2 images and discard 2 (12 fps). - 3: process 3 images and discard 1 (18 fps). - 4: process 4 images and discard 4 (24 fps)

Table 7: Ear detector API

Api for Eye detector

Developers can use the filter's API for its setup. Then, the API is shown:

Function	Description
void viewEyes (int);	To show or hide the bounding boxes of the detected eyes within the image. Parameter's value: - 0 (default), the bounding boxes will not be shown. - > 0, the bounding boxes will be drawn in the frame.
void detectByEvent (int);	To indicate to the algorithm if it must process all the images or only when it receives a specific event such as face detection. Parameter's value:

	- 0 (default), process all the frames; - 1, process a frame when it receives a specific event
void sendMetaData (int);	To send the bounding boxes of the eyes detected to another ME as a metadata. Parameter's value: - 0 (default), metadata are not sent. - 1, metadata are sent.
void multiScaleFactor(int);	To generate the cascade of classifier, the algorithm creates a pyramid iteration scale, in every iteration the image is reduced. Through this method, the user can specify how much the image size is reduced at each image scale. A higher value the processing time will be reduced and the robustness of the algorithm will decrease. A good parameter value will be 20. The value by default is 25.
void withToProcess(int)	This will be the width of the image that the algorithm is going to process to detect eyes. A higher value produces better results but the processing time will be higher. A good parameter value will be 320.
void processXevery4Frames(int)	Through this method we indicate to the algorithm to process x frames every 4 frames. The x value can be: - 0: process 0 images and discard 4. It is like working at 0 fps. - 1: process 1 image and discard 3 (8 fps). - 2: process 2 images and discard 2 (12 fps). - 3: process 3 images and discard 1 (18 fps). - 4: process 4 images and discard 4 (24 fps)

Table 8: Eye detector API

5.4.2.1.3 Capabilities

The idea of this section is to show the capabilities or results that the modules provide to the external world. For this reason we are going to see the modules as a black box, seeing its inputs, its outputs and some particularities.

Inputs and outputs of the Face Detector

As all the rest of VCA filters, this one will receive a stream of images as input. The output of the filter will be a collection of bounding box. Each bounding box represents the position of each face in the image. A bounding box is an area defined by two points. It is very important to highlight that this algorithm **only detects frontal faces**. Therefore, all the faces that are laterally focused will not be detected.

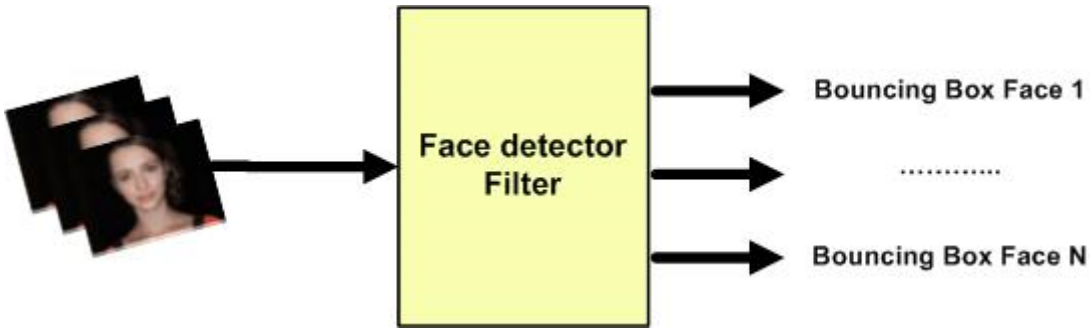

Figure 68: Face Detector Filter Input and Outputs.

Inputs and outputs of the Nose Detector

This filter will also receive a stream of images as input. The output of the filter will be a collection of bounding box. Each bounding box represents the position of each nose found in the image.

But this algorithm needs to detect previously the different faces included on the image. It can detect the faces by its own or can receive the bounding box of the faces included on the image as an input. Therefore, if the filter receives the bounding box of the faces as an input, it will not be necessary to detect the faces again. As a consequence, the processing time of the filter will be reduced. Once the faces have been detected, it will proceed to detect a nose in those parts of the image where there is a face. Moreover, all the bounding boxes representing the faces can be reduced by 20 to 30 percent the height of said regions as both the top and the bottom of the bounding box. In this way, we remove not only information which is not useful for detecting noses as the chin and forehead, but also we can reduce processing time and improve the likelihood of success of the algorithm.

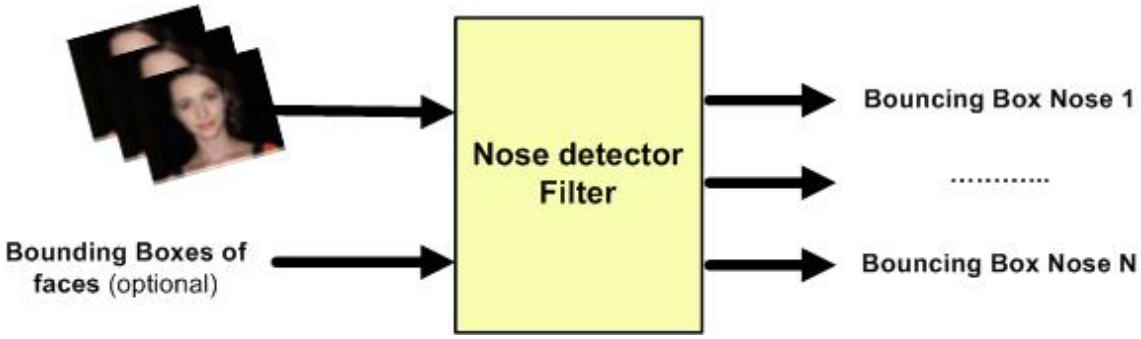

Figure 69: Nose Detector Filter Input and Outputs.

Inputs and outputs of the Mouth Detector

This filter will also receive a stream of images as input. The output of the filter will be a collection of bounding box. Each bounding box represents the position of each mouth found in the image.

In the same way as the nose detector algorithm, this algorithm needs to detect previously the different faces included on the image. This filter can detect the faces by its own or can receive the bounding box of the faces included on the image as an input. Therefore, if the filter receives the bounding box of the faces as an input, it will not be necessary to detect the faces again. As a consequence, the processing time of the filter will be reduced. Once the faces have been detected, it will proceed to detect a mouth in

those parts of the image where there is a face. Moreover, all the bounding boxes representing the faces can be reduced by approximately 50 percent the height of said regions to the top. In this way, we remove not only information which is not useful for detecting mouths as forehead and part of the nose, but also we can reduce processing time and improve the likelihood of success of the algorithm.

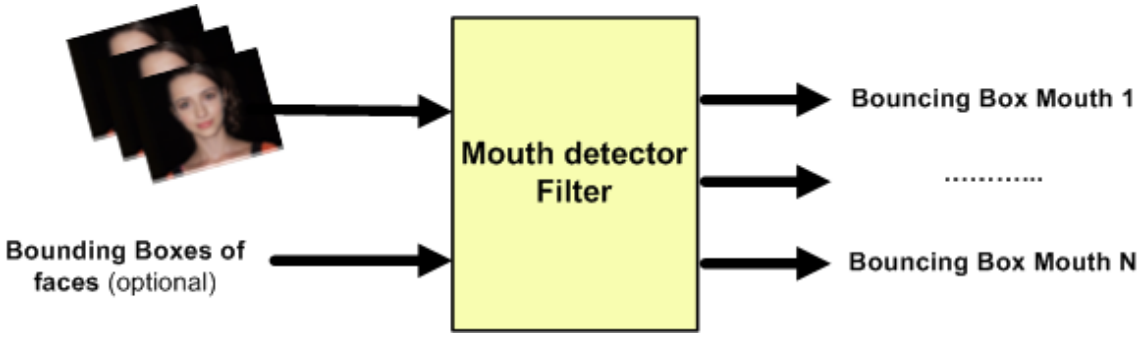

Figure 70: Mouth detector filter Inputs and Outputs

Inputs and outputs of the Ear Detector

This filter will also receive a stream of images as input. The output of the filter will be a collection of bounding box. Each bounding box represents the position of each ear found in the image.

In the same way as the nose and mouth detector algorithm, this algorithm needs to detect previously the different faces included on the image. But in this case, the faces find in the image must be profile faces and not front ones. Therefore, this filter need to detect profile faces by its own, since there is not going to be any filter which detects profile faces. Once the profile faces have been detected, it will proceed to detect an ear in those parts of the image where there is a profile face. Moreover, all the bounding boxes representing the profile faces can be reduced by approximately 30 percent the height of said regions as both the top and the bottom of the bounding box. In this way, we remove not only information which is not useful for detecting ears as chin and forehead, but also we can reduce processing time and improve the likelihood of success of the algorithm.

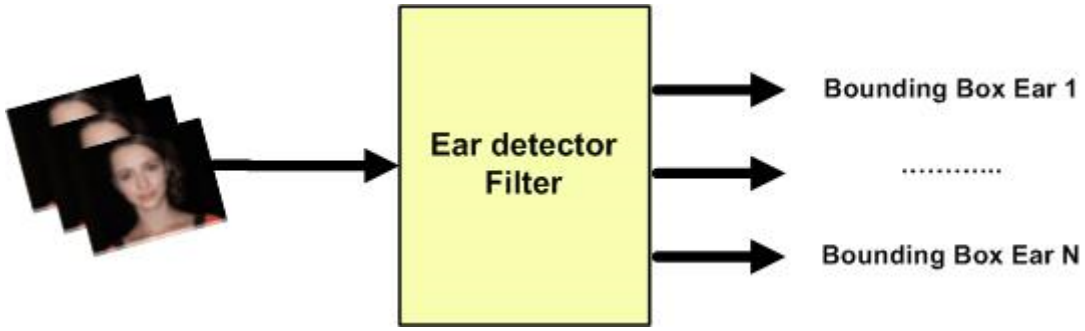

Figure 71: Ear detector filter Input and Outputs

Inputs and outputs of the Eye Detector

This filter will also receive a stream of images as input. Additionally, the algorithm can receive as input the positions of the faces found. The output of the filter will be a collection of bounding box. Each bounding box represents the position of each eye found in the image.

In the same way as some of the others algorithms which detect other face's segments, such as mouth and nose detector, this algorithm needs to detect previously the different faces included on the image. This filter can detect the faces by its own or can receive the bounding boxes of the faces included on the image as an input. Therefore, if the filter receives the bounding box of the faces as an input, it will not be necessary to detect the faces again. As a consequence, the processing time of the filter will be reduced. Once the faces have been detected, it will proceed to detect the eyes in those parts of the image where there is a face. Moreover, all the bounding boxes representing the faces can be reduced by approximately 60 percent the height of said regions. In this way, we remove not only information which is not useful for detecting eyes as the chin and forehead, but also, we can reduce processing time and improve the likelihood of success of the algorithm. It is important to highlight that as the algorithm is looking at each detected face two eyes, therefore we need to call twice the OpenCV function responsible for creating the cascade of classifiers with different parameters. This fact implies an increase in processing time. With the aim to minimize the processing time, despite the 60 percent of the face's height, we split this segment of the face in two in order to minimize the processing time spent searching for eyes, for each function call.

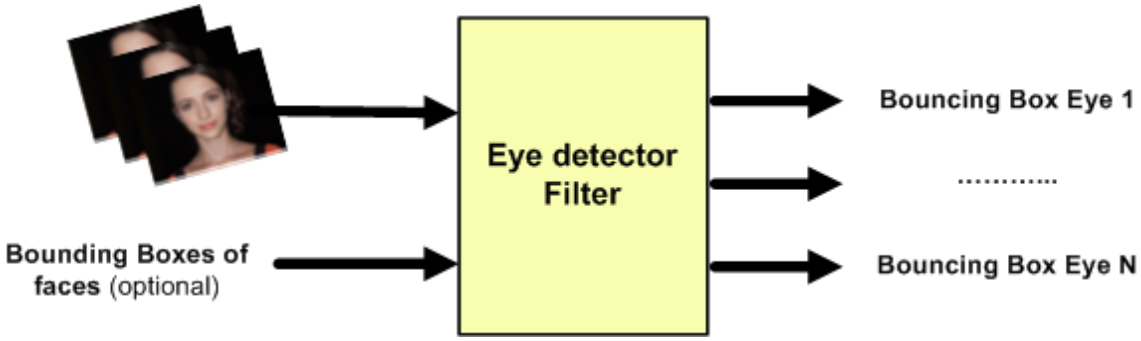

Figure 72: Eye detector filter inputs and outputs

5.4.2.1.4 Evaluation and validation

In this section, we are going to see some metrics in order to see the performance of each algorithm and demonstrator.

Here, we are going to analyze two different kinds of metrics. On the one hand, we are going to study the time that it takes the algorithms to process an image from receiving the image until they finish the processing. On the other hand, we are going to measure the time that it takes from an image enters the pipeline until it reaches the user.

We are going to start with the measure of the algorithms. First of all, we will see the measures for all these algorithms based on the **Haar cascade classifier**. The measures have been taken considering the following parameters:

- Resolution: this is the resolution of the image that the algorithms process. For that purpose, it is necessary to make different operations such as the resizing of the image when the algorithm receives the image.
- Frame per second: with this parameter, we can avoid or reduce problems related to the cumulative delay of the buffers, when it takes the algorithms to process to much time. In this way, the algorithm can process only 1,2,3 or 4 frames per second of every 4 images.

- Scale Factor: The classifier is designed so that it can be easily "resized" in order to be able to find the objects of interest at different sizes. So, to find an object of an unknown size in the image the scan procedure should be done several times at different scales. Therefore, with this parameter we specify how much the image size is reduced at each image scale.

Figure 73: Haar cascade: scale pyramid

Based on these parameters we are going to see the result obtained for face profile algorithms. In the following table, we can see the results to run the algorithms with difference resolutions. Here, it is important to note that the mouth, nose and eye detector does not make the face detection, previous step to detect each facial part, since they obtain the bounding boxes of the face through the face detector Media Element included in the same pipeline. About the results, we need to remark that when we work at 160x120 px, the mouth, nose, eye and ear detector work very bad obtaining dreadful results. That is why we prefer not to include the processing time of this algorithm for this resolution, and we do not recommend using these algorithms with these resolutions. Furthermore, based on the result we recommend using the face detector with 160x120px and 320x240 px

Alg /res	160x120 px	320x240 px	640x480 px
Face	7 ms	25-30 ms	65-75 ms
Mouth	-	6/8 ms	12 ms
Nose	-	8 ms	15 ms
Eye	-	12 ms	25-30 ms
Ear	-	18 -23 ms	40 ms

Table 9: Benchmark: face segments processing time (resolution)

As for the scale factor, this parameter only has sense to be taken into account for the face detector algorithm. Because as it has been explained in the part corresponding to these algorithms on section 6, once the face has been detected, they search in a reduced part of the image, where the face is located, for the specific face segment. Therefore, the difference for these algorithms is almost negligible.

Scale factor	10 %	20 %	30 %
Face (320x240)	70 ms	33 ms	20 ms
Face (160x120)	12 ms	8 ms	5/6 ms

Table 10: Benchmark: face detection processing time (scale factor)

The other parameter Frame per seconds does not directly affect the performance of each frame, what really makes it is to avoid or reduce problems related to the cumulative delay of the buffers. For example, if we process the face detector algorithm with a resolution of 640x480 px, the video will end with a visible delay for the end user. However, working at 12 fps (discarding 2 images of 4) is likely to eliminate this delay.

Once, we have seen metrics for the processing time of each algorithm, we are going to see the end to end video latency for the different demos created to test this algorithm. With the goal to obtain the end to end video latency, we are using the object Stats provided by the Kurento platform.

Algorithm	Vide e2e lantecy
Face	20 – 22 ms
Mouth*	25 – 28 ms
Nose*	26 – 30 ms
Eye*	28 – 32 ms
Ear*	38 – 41 ms
Face Profile**	70 – 75 ms

Table 11: Benchmark: Video end to end latency for facial segments

* This algorithms also need to run a face detector previous to detect its specific facial segment

** Pipeline composed by Face, Mouth, Nose, and Eye media elements

The values of the metrics of all these tables included in this section are approximate and may vary depending on different aspects. These metrics have been taken with the default configuration and under conditions that we consider normal to use. An aspect that can influence these metrics, for these algorithms, is the distance from the face to the camera. The closer the face of the camera, the greater the surface to detect and the process time will be increased.

Another important issue to comment is that it takes the ear detector a little longer to process than similar algorithms such as Mouth, Nose and Eye. This is due to these algorithms need to detect a frontal face and the ear detector searches to detect the left and right side of the face, thus increasing the computational load.

5.4.2.1.5 Information for developers

Licence

In the following table, you can find the license corresponding with the algorithm explained on this section.

Algorithm	License
Face detector	LGPL v2.1
Nose detector	LGPL v2.1
Mouth Detector	LGPL v2.1

Eye Detector	LGPL v2.1
Ear Detector	LGPL v2.1

Table 12: License for the demos of facial segments

Obtaining the source code

With the aim to obtain the source code of these filters, both modules and demos, you can obtain it through a github repository.

The source code of the modules of these algorithms can be found in the following url:

Algorithm	Git link
NuboFaceDetector	https://github.com/VTOOLS-FOSS/NUBOMEDIA-VCA/tree/master/modules/nubo_face/nubo-face-detector
NuboMouthDetector	https://github.com/VTOOLS-FOSS/NUBOMEDIA-VCA/tree/master/modules/nubo_mouth/nubo-mouth-detector
NuboNoseDetector	https://github.com/VTOOLS-FOSS/NUBOMEDIA-VCA/tree/master/modules/nubo_nose/nubo-nose-detector
NuboEarDetector	https://github.com/VTOOLS-FOSS/NUBOMEDIA-VCA/tree/master/modules/nubo_ear/nubo-ear-detector
NuboEyeDetector	https://github.com/VTOOLS-FOSS/NUBOMEDIA-VCA/tree/master/modules/nubo_eye/nubo-eye-detector

Table 13: Github Repositories for the modules of facial segments

In addition, you can find the demos developed up to now which use the different media elements in the following links:

Demo	Description and Link
NuboFaceJava	*Description*: pipeline created by a face detector https://github.com/VTOOLS-FOSS/NUBOMEDIA-VCA/tree/master/apps/NuboFaceJava
NuboMouthJava	*Description*: pipeline created by a mouth detector https://github.com/VTOOLS-FOSS/NUBOMEDIA-VCA/tree/master/apps/NuboMouthJava
NuboNoseJava	*Description*: pipeline created by a nose detector https://github.com/VTOOLS-FOSS/NUBOMEDIA-VCA/tree/master/apps/NuboNoseJava
NuboEyeJava	*Description*: pipeline created by an eye detector https://github.com/VTOOLS-FOSS/NUBOMEDIA-VCA/tree/master/apps/NuboEyeJava

NuboEarJava	*Description*: pipeline created by an ear detector https://github.com/VTOOLS-FOSS/NUBOMEDIA-VCA/tree/master/apps/NuboEarJava
NuboFaceProfileJava	Description: pipeline created by a face, mouth and nose detector http://80.96.122.50/victor.hidalgo/vca_media_elements/tree/master/apps/NuboFaceProfileJava

Table 14: Github Repositories for the demos of facial segments

5.4.2.2 Motion detection

5.4.2.2.1 Rationale and objectives

This algorithm has been selected to be included on the NUBOMEDIA platform for the following reasons:

- The main reason to include this algorithm on NUBOMEDIA project is a strategic decision. Since this is a typical functionality demanded by our customer, we are interested in using this algorithm on a cloud infrastructure where the filter can be combined with others to explore the benefits this algorithm can bring to customers.
- The core of the technology has already been developed, and only some minor changes have been added to the filter. This has allowed us to be agile in the generation of the filter

The main goal of this algorithm is to detect motion in the image. This algorithm can be used for different uses. A typical use is to store the sequences of video when motion has been detected on the image. In this way, the users can reduce considerably the size of stored information. Another example can be the search of video sequences, seeking only those parts of the video where motion is detected. Thus, we can save considerable time when we search a specific video sequence.

5.4.2.2.2 Software Architecture

For this algorithm, we are using a Visual Tools' proprietary software. Basically, this algorithm is based on an image difference. The image difference consists of subtracting one by one the corresponding pixels between two images. The difference of pixels is done by subtracting the pixel values, such as RGB values, in absolute value. If the result is higher than a specific threshold, then the result for that pixel is that we have detected motion. In the following image you can find an example of image subtraction

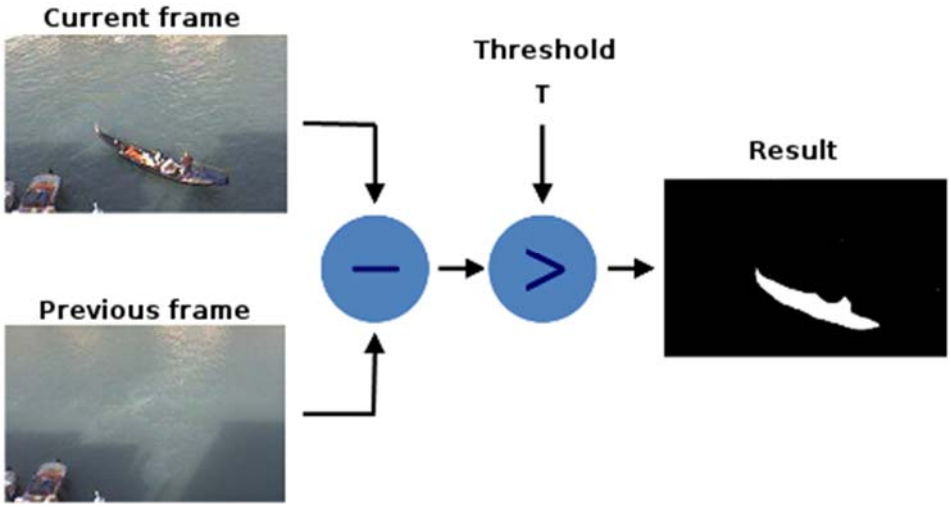

Figure 74: Motion detection: Image difference

This filter splits the images in a matrix of 10x8 blocks and weight up the changes that happen in each block between two consecutive frames. At the end of the processing, if the number of blocks where has been detected movement is higher than a specific threshold, the conclusion of the algorithm is that there has been movement. This threshold is different than the threshold used to make the image difference and indicates the minimum number of blocks where motion has been detected.

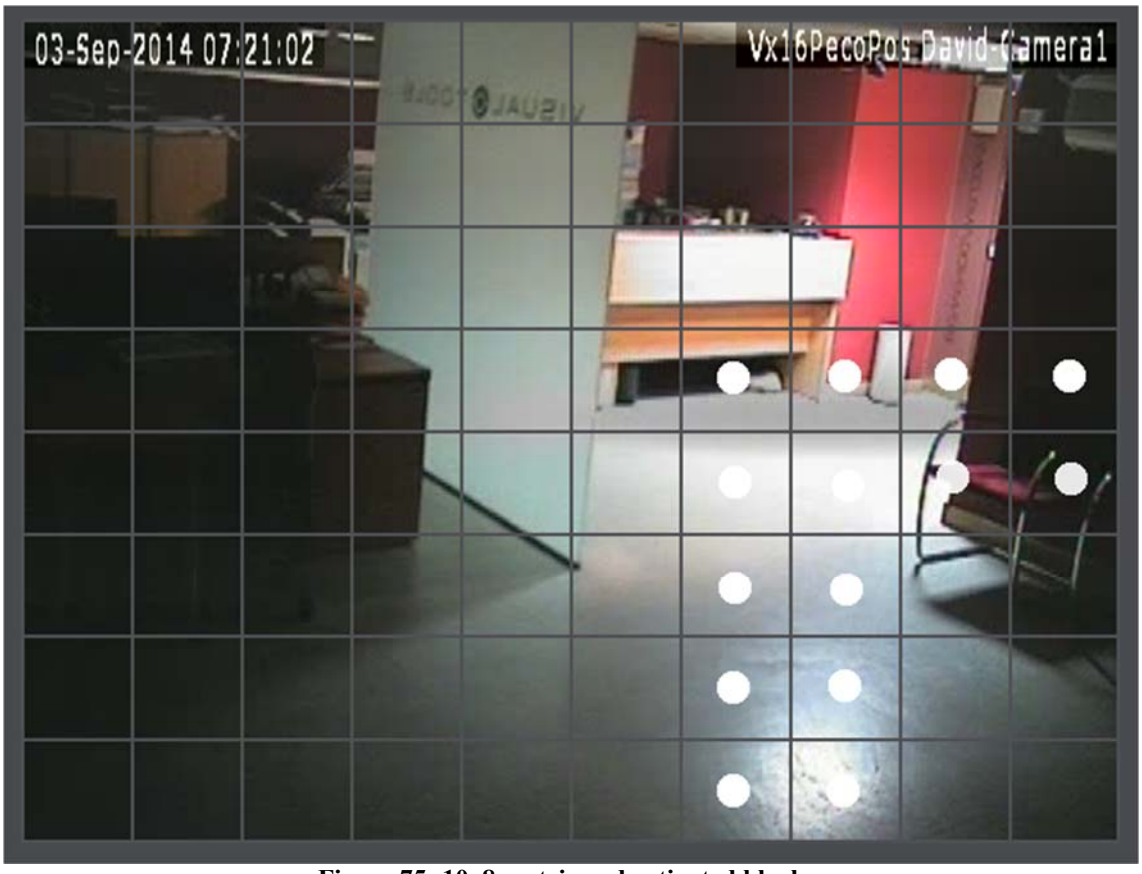

Figure 75: 10x8 matris and activated blocks

Apart from this threshold, it is also configurable the blocks of the matrix which are going to be processed. If you look at the previous image, you will see an image with the 10x8 blocks corresponding to the matrix. You can also observe some white points in

some specific blocks. These points indicate the blocks where the algorithm will carry out the processing.

Once we have seen the main principles for this algorithm, we will see its API.

Api for Motion detector

Developers can use the filter's API for its setup. Then, the API is shown:

Function	Description
void setMinNumBlocks(String);	To set the minimum number the blocks of the grid where the algorithm must detect movement to conclude that there has been motion in the image. Parameter's value: - "high", detect movement in 1 block at least - "medium", detect movement in 3 blocks at least - "low", detect movement in 9 blocks at least
void setGrid (String);	To set the areas (grid) where the algorithm will detect motion. Parameter's value: - The string will be composed by 80 characters. Each character is matched to a block of the grid. The value of character can be 0 (It does not matter to detect motion at this block) or 1 (it matters to detect motion at this block).
void applyConf ();	After set up the minimum number of blocks and set the grid, we need to call this function to make the changes effective
void sendMetaData (int);	To send and event to indicate to another ME that it has been detected movement in the image. Parameter's value: - 0 (default), metadata are not sent. - 1, metadata are sent.

Figure 76: Motion detector API

5.4.2.2.3 Capabilities

The idea of this section is to show the capabilities or results that the modules provide to the external world. For this reason we are going to see the modules as a black box, seeing its inputs, its outputs and some particularities.

This filter will receive a stream of images as input. The output of the filter will be a 1 or a 0. The number 1 indicates that there has been motion detection. Otherwise, if there has not detected any motion, the output of the filter will be 0. It is very important to highlight that this algorithm works very well for **indoors**, while outdoors, the result may be inconsistent.

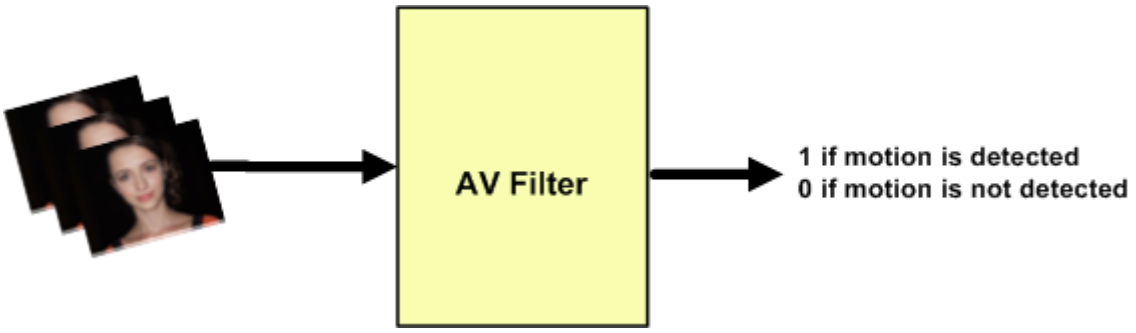

Figure 77: Motion detection Filter Input and Outputs.

5.4.2.2.4 Evaluation and validation

In this section, we are going to see some metrics in order to see the performance of this algorithm and demonstrator.

As it was done in the previous section, we are going to analyze two different kinds of metrics:

- the time that it takes the algorithm to process an image from receiving the image until they finish the processing.
- the time that it takes from an image enters the pipeline until it reaches the end of the pipeline.

As opposed to the filters based on the detection of facial segments, we do not have any particular parameter which can improve the performance of the motion detection algorithm. It always works with the same resolution and does not have the scale factor parameter. In the following table we can see the processing time.

Algorithm	Processing time
Motion	5 - 7 ms

Table 15: Benchmark: Motion detection processing time

In the following table, we can see the end to end video latency for the demo created for the motion detector. For this demo, we need to highlight that the pipeline has been composed by two different filters, the motion and face detector.

Algorithm	Vide e2e lantecy
Motion + Face	40 ms

Table 16: Benchmark: Motion detection video end to end latency

In the same way as the filters based on the detection of facial segments, the values of the metrics are approximate and may vary depending on different aspects. These metrics have been taken under conditions that we consider normal to use. In this particular case, the video end to end latency can be higher, when there is more movement in the image,

5.4.2.2.5 Information for developers

Licence

As for the license, the motion detector algorithm is **proprietary software**.

Obtaining the source code

Therefore, the source of the algorithm will not be provided. This software will be only used with the consent of the owner, in this case Visual Tools. The binaries will be provided to the project consortia for the duration of the project,

However, the source code of the demo (web interface) is available for all the partners of the consortia. They can access to the source code in the following url:

Demo	Description and Link
NuboMotionFaceJava	*Description:* pipeline created by a motion and Face detector http://80.96.122.50/victor.hidalgo/vca_media_elements/tree/master/apps/NuboMotionFaceJava

Table 17: Repository of the motion and face demo

5.4.2.3 Virtual Fence

5.4.2.3.1 Rationale and objectives

This algorithm has been selected to be included on the NUBOMEDIA platform for the following reasons:

- The main reason to include this algorithm on NUBOMEDIA project is a strategic decision. Since this is functionality demanded by our customer. This functionality is not as demanded as the motion detection because it is a more specific solution that our customers ask us to turnkey projects, particularly for securing power solar plants, or high-security building. And in the same way as the previous algorithm, we are interested in using this algorithm on a cloud infrastructure where the filter can be combined with others to explore the benefits this algorithm can bring to customers.
- The core of the technology has already been developed, and we have only introduced some minor changes to the filter. This has allowed us to be more agile in the generation of the filter. However, in this case the integration of the module in the NUBOMEDIA platform has been quite hard, due to the complexity of the algorithm.

The Virtual Fence algorithm is a security application used to avoid access to a restricted area. The aim is to generate an alarm if an object crosses certain virtual line. A good example of using this algorithm can be in a prison. In this environment, prisoners can try to escape. In the following figure, we can see how a prisoner is trying to escape. In

this case, when the prisoner crosses the red virtual line an alarm will be generated.

Figure 78: Virtual Fence

5.4.2.3.2 Software Architecture

The Virtual fence algorithm is based on a corner detector or high curvature points algorithm [HARRIS1998]. A corner detector algorithm considers a local window in the image, and determining the average changes of image intensity. Therefore, a corner can be defined as the features of a digital image that correspond to the points with high spatial gradient and high curvature or in other words a corner is the intersection of two edges. For example, a point in the middle of a table is a bad option to be a corner, since there is no a big change of texture or intensity. However, a point in the corner of a table is an interesting point for the algorithm since there is a big change of texture or intensity between the table and probably the floor. In the following figure we can see on the left side the original image and on the right side the image with the corners (red points) calculated.

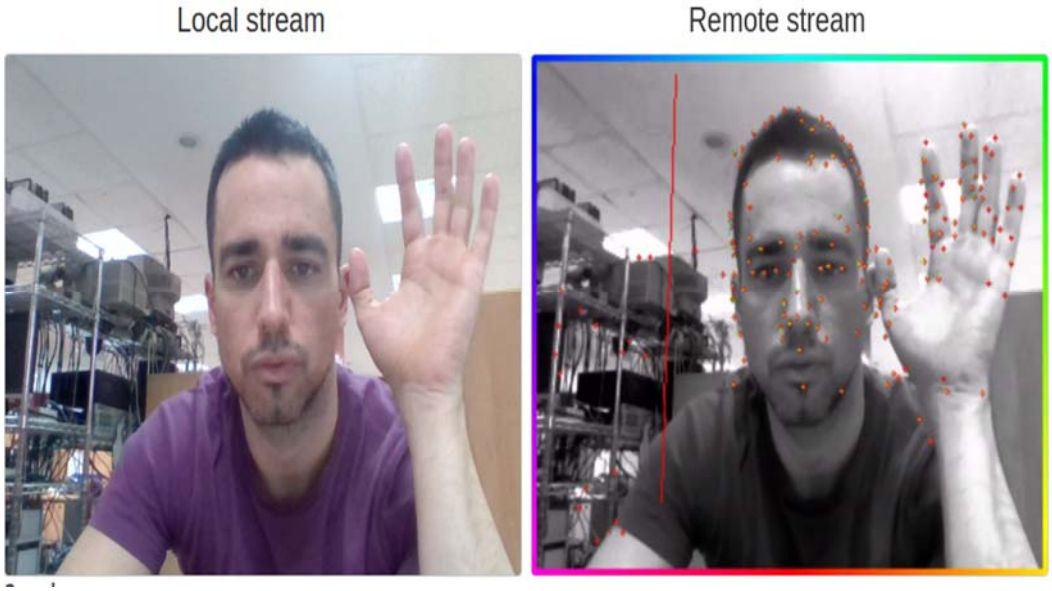

Figure 79: Virtual Fence: Corners

Once the algorithm detects a corner in a frame, it tries to find exactly the same point in the previous frame. In this way, making the matching of a point in a frame and the previous frame, we can follow points over time. In addition, when a number of points, which are configurable, cross the virtual line the system will launch an alarm, indicating

that an intrusion has taken place. In the following figure, we can see the trajectory of the different corners. The color of this trajectory, given by the picture frame, indicates the direction of the corners. In this example, when an intrusion is detected the picture frame becomes red, indicating that an alarm has been launched.

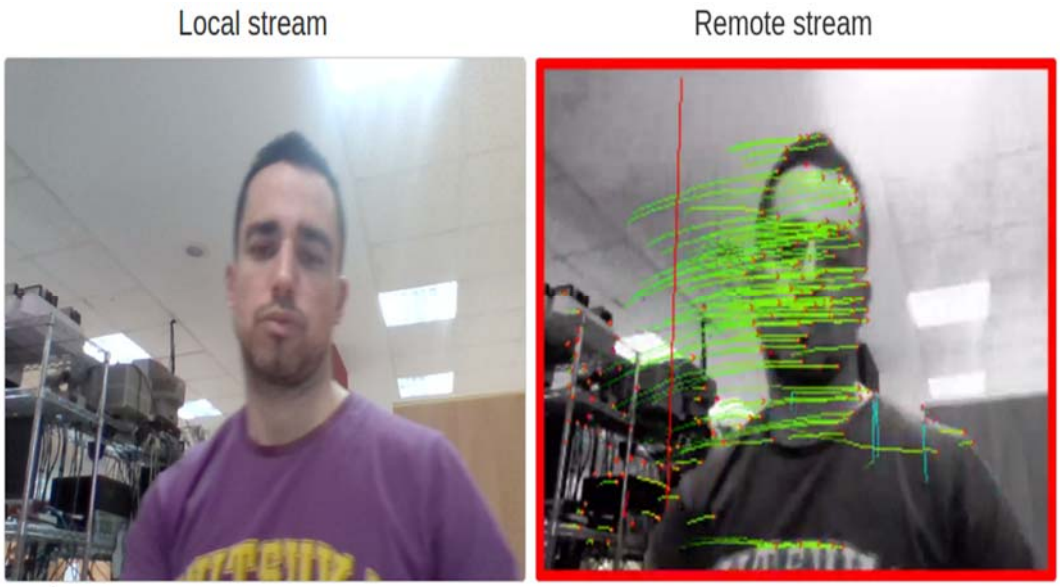

Figure 80: Virtual Fence: Trajectory of the corners

Another important characteristic of the corners is that if the position of a specific corner over the time is stored you can study the trajectory of them. This fact is very important, because if we cannot study the trajectory of the points a lot of false alarms are going to be generated. When you install this system on real environments, you face to problems that you cannot find in a lab. In this case, we have found many problems due to insects and cobwebs mainly at nights. In the following figure, we can see an example of these problems.

Figure 81: Virtual Fence: false alarms caused by insects and cobwebs.

Api for Virtual Fence

Developers can use the filter's API for its setup. Then, the API is shown:

Function	Description
void viewVfences (int);	To show or hide the virtual lines and corners calculated by the algorithm within the image. Parameter's value:

	- 0 (default), the fences and corners will not be shown. - > 0, the fences and corners will be shown.
void setFences (String);	To set up the virtual line. The Parameter is a String with the following format: - "*p1_x, p1_y, p2_x, p2_y*" Where **p1_x, p1_y** are the coordinates of the first point of the line, and **p2_x, p2_y** are the coordinates of the second point of the line.
void setThreshold (int);	To set up the minimum number of corners that should cross a line to consider an intrussion. Parameter's value: - 0 – 100 (6 default)
void applyConf();	To apply the new configuration established by setThreshold and setFences .
void sendMetaData (int);	To send and event to indicate to another ME that an intrusion has been detected. Parameter's value: - 0 (default), metadata are not sent. - 1, metadata are sent.

Figure 82: Virtual Fence API

5.4.2.3.3 Capabilities

Now we are going to see the capabilities or results that the modules provide to the external world. For this reason we are going to see the modules as a black box, seeing its inputs, its outputs and some particularities.

This filter will receive a stream of images as input. The output of the filter will be an event as long as the algorithm will detect an intrusion. In the following figure you can find a graphical representation about the inputs and outputs of the algorithm.

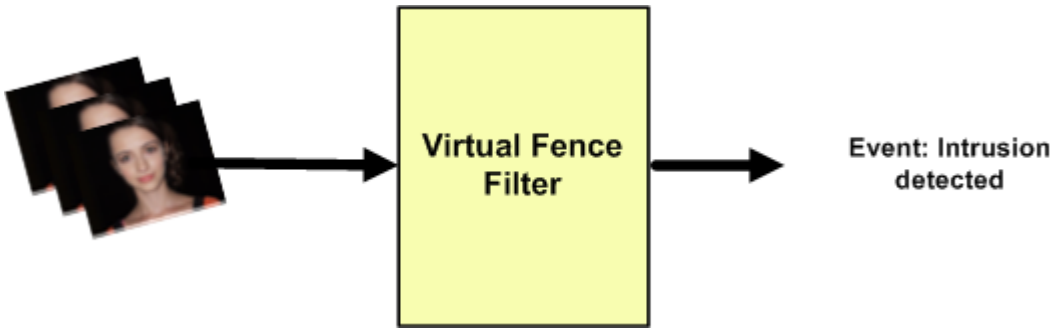

Figure 83: Virtual Fence Inputs and Outputs

5.4.2.3.4 Evaluation and validation

In this section, we are going to see some metrics in order to see the performance of this algorithm and its demonstrator.

As it was done in the previous sections, we are going to analyze two different kinds of metrics:

- the time that it takes the algorithm to process an image from receiving the image until they finish the processing.
- the time that it takes from an image enters the pipeline until it reaches the end of the pipeline.

As opposed to the filters based on the detection of facial segments, we do not have any particular parameter which can improve the performance of the virtual fence algorithm. It always works with the same resolution and does not have the scale factor parameter. In the following table we can see the processing time.

Algorithm	Processing time
Virtual Fence	18 - 23 ms

Table 18: Benchmark: Virtual Fence processing time

In the following table, we can see the end to end video latency for the demo created for this filter.

Algorithm	Vide e2e lantecy
Motion + Face	40 ms

Table 19: Benchmark: Virtual Fence video end to end latency

In the same way as the previous filters, the values of the metrics are approximate and may vary depending on different aspects. These metrics have been taken under conditions that we consider normal to use. In this particular case, the processing time depends on the movement in the picture and the number of corners.

5.4.2.3.5 Information for developers

License

As for the license, the motion detector algorithm is **proprietary software**.

Obtaining the source code

Therefore, the source will not be provided. This software will be only used with the consent of the owner, in this case Visual Tools. The binaries will be provided to the project consortia for the duration of the project,

However, the source code of the demo (web interface) is available for all the partners of the consortia. They can access to the source code in the following url:

Demo	Description and Link
NuboVfenceJava	*Description:* pipeline created by the Virtual Fence filter http://80.96.122.50/victor.hidalgo/vca_media_elements/tree/master/apps/NuboVfenceJava

Figure 84: Repository of the virtual fence demo

5.4.2.4 Tracker

5.4.2.4.1 Rationale and objectives

These algorithms have been selected to be included on the NUBOMEDIA platform for the following reasons:

- They are an algorithm which is widely used in industrial applications, and we believe that it could be of great use for the NUBOMEDIA partners and possible future users of the platform.
- Four partners in the project are interested in it.
- There is lot information about the tracking techniques based on computer vision which allows us to develop them in reasonable time.

The main goal of this algorithm is to detect and follow objects. In the following section we will see more characteristics about it

5.4.2.4.2 Software Architecture

The first important thing that we need to take into account is that this algorithm implements a tracking system based on motion detection and no other techniques such as detection of colors. According to Wikipedia, '*Motion Detection is a process of detecting a change in the position of an object relative to its surroundings*'.

The algorithm that we have developed is based on OpenCV and follows these steps:

- Image difference
- Motion History
- Motion Gradient
- Segment Motion

However, previous to start with the steps is necessary to make a pre-processing of the image. In this case, we only need to convert the image to a gray format. As we have explained before, this algorithm tries to track objects based on motion, therefore images in a specific color format will be heavier. As a consequence, processing each image in each of these steps will be slower. That is why the first operation previous to the different steps is to convert the image to a gray format. Now, let's see each step separately.

Image difference

Through this first step, the main idea is to make the absolute difference between to consecutives frames, with the aim to get the silhouettes of moving objects. A part from the image difference, this step includes the threshold operation. Once, we have done the image difference we apply the thresholding operation. The thresholding is used to segment an image by setting all pixels whose intensity values are above a threshold to a foreground value and all the remaining pixels to a background value. The result of this steps can be seen in the following figure:

Figure 85: Tracker: image difference

Motion history

The concept of Motion History Image (MHI) forms the basis of Motion Detection. Let us assume that an object silhouette is created which is most recent. A 'timestamp' is used to identify if the image is recent or not, which is basically the current system time, measured upto milliseconds. The other older silhouettes are necessary to be compared with this silhouette in order to achieve motion detection. Hence, older silhouettes are also saved in the image, with an older timestamp. We can say that the sequence of these silhouettes along with the timestamp, which record the previous motion, is referred to as the "Motion History Image".

Figure 86: Tracker: motion history

Motion Gradient

Once the motion template has a collection of object silhouettes overlaid in time, we can derive an indication of overall motion by taking the gradient of the mhi image. But first of all let's understand what a gradient is.

In mathematics, the gradient is a generalization of the usual concept of derivative of a function in one dimension to a function in several dimensions. f f(x1, ..., xn) is

a differentiable, scalar-valued function of standard Cartesian coordinates in Euclidean space, its gradient is the vector whose components are the partial derivatives of f. It is thus a vector-valued function. Similarly to the usual derivative, the gradient represents the slope of the tangent of the graph of the function. More precisely, the gradient points in the direction of the greatest rate of increase of the function and its magnitude is the slope of the graph in that direction. The components of the gradient in coordinates are the coefficients of the variables in the equation of the tangent space to the graph. This characterizing property of the gradient allows it to be defined independently of a choice of coordinate system, as a vector field whose components in a coordinate system will transform when going from one coordinate system to another. We are going to see an example to illustrate better this concept. Imagine a room in which the temperature is given by a scalar field, T, so at each point (x, y, z) the temperature is T(x, y, z). (assume that the temperature does not change over time.) At each point in the room, the gradient of T at that point will show the direction the temperature rises most quickly. The magnitude of the gradient will determine how fast the temperature rises in that direction. In the above two images, the values of the function are represented in black and white, black representing higher values of temperatures, and its corresponding gradient is represented by blue arrows.

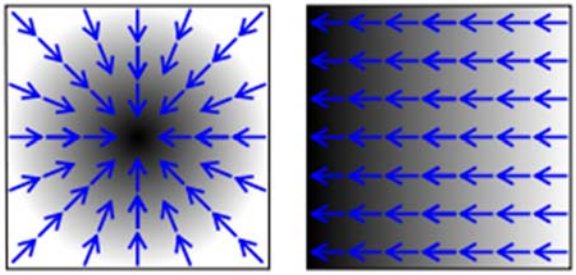

Figure 87: Tracker: gradient

In our case we will use the motion history image in order to calculate the gradient.

Segment motion

Once the gradient has been calculated, we split a motion history image into a few parts corresponding to separate independent motions (for example, left hand, right hand).

Once all this parts have been carried out, we detect the objects based on its corresponding motion. A part from this process, the algorithm can behave in a different way depending on the environment where it runs. For example, initially this algorithm works better for outdoors where the objects are far from the camera. You can find an example on the left part of the following image how the result is not as expected when the camera is near the object. What really happens is that the silhouette history of the moving object is not continuous, as we can see on figure 2. We can see the result in left side of the following image. However, applying the right configuration we get a good result as we can see on the right side of this image. The configuration's parameters are the following:

- Threshold: To set up the minimum difference among pixels to consider motion
- Min Area: To set up the minimum area to consider objects.
- Max Area: To set up the maximum area to consider objects.
- Distance: To set up the distance among objects to merge them.

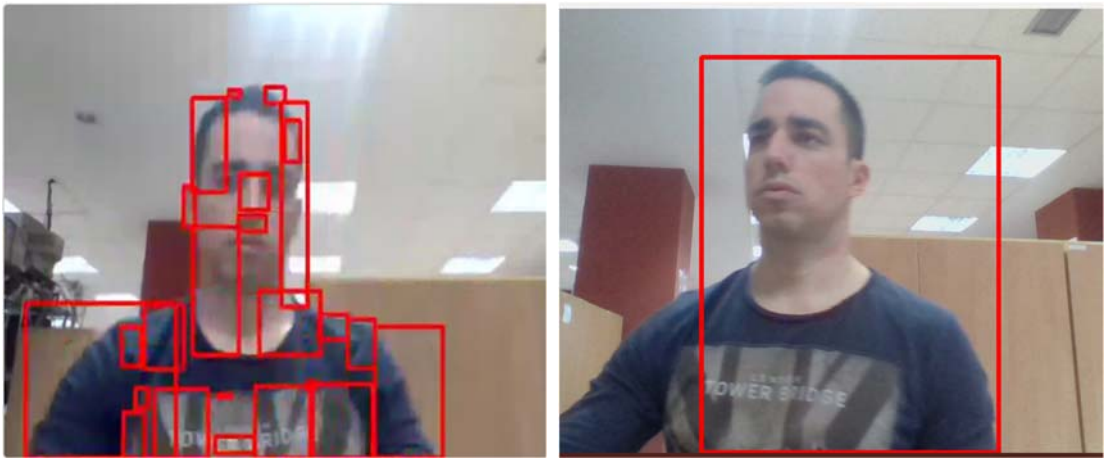

Figure 88: Tracker final result

Api for Tracker

Developers can use the filter's API for its setup. Then, the API is shown:

Function	Description
void setVisualMode (int);	To show or hide the objects detected. Parameter's value: - 0 (default), the bounding boxes will not be shown. - 1, the bounding boxes will be drawn in the frame.
void setThreshold (int);	To set up the minimum difference among pixels to consider motion Parameter's value: - 0-255 (20 default)
void setMinArea (int);	To set up the minimum area to consider objects. Parameter's value: - 0 - 10000 (50 default)
void setMaxArea(int);	To set up the maximum area to consider objects. Parameter's value: - 0 - 300000 (30000 default)
void setDistance (int);	To set up the distance among objects to merge them. Parameter's value: - 0 – 2000 (35 default)

Figure 89: Tracker API

5.4.2.4.3 Capabilities

The idea of this section is to show the capabilities or results that the module provides to the external world. For this reason, we are going to see the module as a black box, seeing its inputs, its outputs and some particularities.

This filter will receive a stream of images as input. The output of the filter will be the bounding boxes of the objects found in the image. In the following figure, you can find a graphical representation about the inputs and outputs of the algorithm.

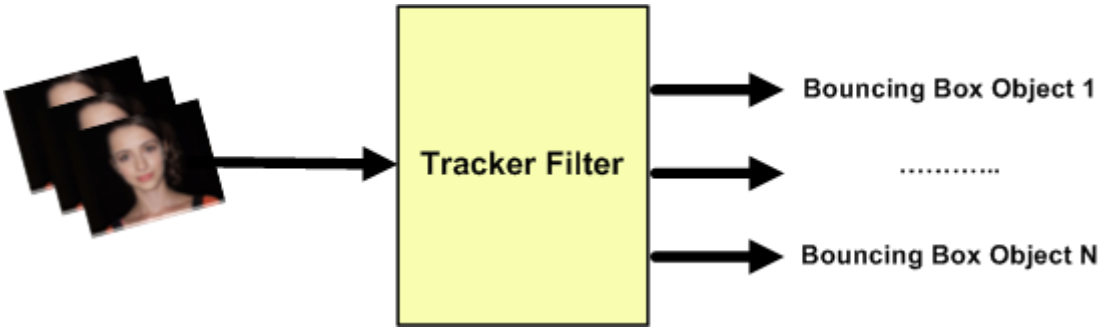

Figure 90: Tracker Filter Input and Outputs.

5.4.2.4.4 Evaluation and validation

In this section, we are going to see some metrics in order to see the performance of this algorithm and demonstrator.

As it was done in the previous section, we are going to analyze two different kinds of metrics:

- the time that it takes the algorithm to process an image from receiving the image until they finish the processing.
- the time that it takes from an image enters the pipeline until it reaches the end of the pipeline.

In the following table, we can see the processing time.

Algorithm	Processing time
Tracker	10 - 15 ms

Table 20: Benchmark: Motion detection processing time

In the following table, we can see the end to end video latency for the demo created for the Tracker. F

Algorithm	Vide e2e lantecy
Tracker	25 ms

Table 21: Benchmark: Motion detection video end to end latency

In the same way as other filters, the values of the metrics are approximate and may vary depending on different aspects. These metrics have been taken under conditions that we consider normal to use. In this case, the processing can be higher, when there is more movement in the image,

5.4.2.4.5 Information for developers

This algorithm has been released under the LGPLv2.1 license.

Obtaining the source code

With the aim to obtain the source code of this filter, both module and demo, you can obtain it through a github repository.

The source code of the module can be found in the following url:

Algorithm	Git link
NuboTracker	https://github.com/VTOOLS-FOSS/NUBOMEDIA-VCA/tree/master/modules/nubo_tracker/nubo-tracker

Figure 91: Github Repositories for the Tracker module

In addition, you can find the demo developed in the following links:

Demo	Description and Link
NuboTrackerJava	*Description:* pipeline created by a Tracker filter https://github.com/VTOOLS-FOSS/NUBOMEDIA-VCA/tree/master/apps/NuboTrackerJava

Figure 92: Github Repositories for the Tracker demo

5.4.3 Installation guide for VCA software

Now it is described the steps performed to install the VCA filters and demos developed in the images that will deploy on the NUBOMEDIA cloud. This section is divided into two main parts. On the one hand, we will see how to install the filters using the repositories of the project. On the other hand, we will see how to install both filters and demos from source code.

- **Previous steps.**

Before beginning these parts, you need to install the following packages:

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

- **Installing filter from NUBOMEDIA repositories**

In order to install the latest stable version of the NUBOMEDIA-vca filters, you have type the following commands, one at time and in the same order as listed here. When asked for any kind of confirmation, reply affirmatively:

```
$ apt-key adv --keyserver keyserver.ubuntu.com --recv-keys F04B5A6F
$ add-apt-repository "deb http://repository.nubomedia.eu/ trusty main"
$ apt-get update -y
$ apt-get install nubo-face-detector nubo-face-detector-dev
nubo-mouth-detector nubo-mouth-detector-dev nubo-nose-detector
nubo-nose-detector-dev nubo-eye-detector nubo-eye-detector-dev
nubo-tracker nubo-tracker-dev  nubo-ear-detector nubo-ear-detector-dev -y
```

- **Installing filter from NUBOMEDIA from the github**

In order to install the available software from the source code in github, you have to follow the next steps. The partners of the NUBOMEDIA consortium could also download the demos of the proprietary software from the NUBOMEDIA git repository and the deb packages of the private modules will be available for them.

First of all, we need to Clone the repo:

```
$ git clone https://github.com/VictorHidalgoVT/NUBOMEDIA-VCA.git nubo-vca
```

After Cloning the repo, is time to compile, follow these instructions

```
$ cd nubo-vca/
$ sh build.sh
```

If everything goes well, a new folder with the following content has been generated

```
output/
   ├─── apps
   │     ├─── NuboEarJava.zip
   │     ├─── NuboEyeJava.zip
   │     ├─── NuboFaceJava.zip
   │     ├─── NuboFaceProfileJava.zip
   │     ├─── NuboMouthJava.zip
   │     ├─── NuboNoseJava.zip
   │     └─── NuboTrackerJava.zip
   └─── packages
        ├─── nubo-ear-detector_0.0.4~rc1_amd64.deb
        ├─── nubo-ear-detector-dev_0.0.4~rc1_amd64.deb
        ├─── nubo-eye-detector_0.0.4~rc1_amd64.deb
        ├─── nubo-eye-detector-dev_0.0.4~rc1_amd64.deb
        ├─── nubo-face-detector_0.0.4~rc1_amd64.deb
        ├─── nubo-face-detector-dev_0.0.4~rc1_amd64.deb
        ├─── nubo-mouth-detector_0.0.4~rc1_amd64.deb
        ├─── nubo-mouth-detector-dev_0.0.4~rc1_amd64.deb
        ├─── nubo-nose-detector_0.0.4~rc1_amd64.deb
        ├─── nubo-nose-detector-dev_0.0.4~rc1_amd64.deb
        ├─── nubo-tracker_0.0.4~rc1_amd64.deb
        └─── nubo-tracker-dev_0.0.4~rc1_amd64.deb
```

For install the filters, you need to run the following commands:

```
$ cd output/packages
$ sudo dpkg -i nubo-ear-detector_0.0.4~rc1_amd64.deb
nubo-ear-detector-dev_0.0.4~rc1_amd64.deb nubo-eye-setector_0.0.4~rc1_amd64.deb
nubo-eye-detector-dev_0.0.4~rc1_amd64.deb  nubo-face-detector_0.0.4~rc1_amd64.deb
nubo-face-detector-dev_0.0.4~rc1_amd64.deb nubo-mouth-detector_0.0.4~rc1_amd64.deb
nubo-mouth-detector-dev_0.0.4~rc1_amd64.deb nubo-nose-detector_0.0.4~rc1_amd64.deb
nubo-nose-detector-dev_0.0.4~rc1_amd64.deb nubo-tracker_0.0.4~rc1_amd64.deb nubo-
tracker-dev_0.0.4~rc1_amd64.deb
```

For install the demos, you need to run the following commands for every zip file contained in the **output/apps** folder. We will make the example for the face detector.

```
$ cd output/apps
$ mkdir face
$ mv NuboFaceJava.zip face/
$ unzip -x NuboFaceJava.zip
$ sudo sh install.sh
```

- **Running the demos**

To run the difference demos, you need to access the following url's through a web browser compliant with WebRTC.:

- Face Detector => http://localhost:8100
- Motion + Face Detector => http://localhost:8101
- Nose Detector => http://localhost:8102
- Mouth Detector => http://localhost:8103
- Ear Detector => http://localhost:8104
- Face + Nose + eye + Mouth Detector => http://localhost:8105
- Virtual Fence => http://localhost:8106
- Tracker => http://localhost:8107

The following figures show an example of the pipeline created by the Face, Nose, eye and Mouth detector (face profile) demo, and the proper demo running.

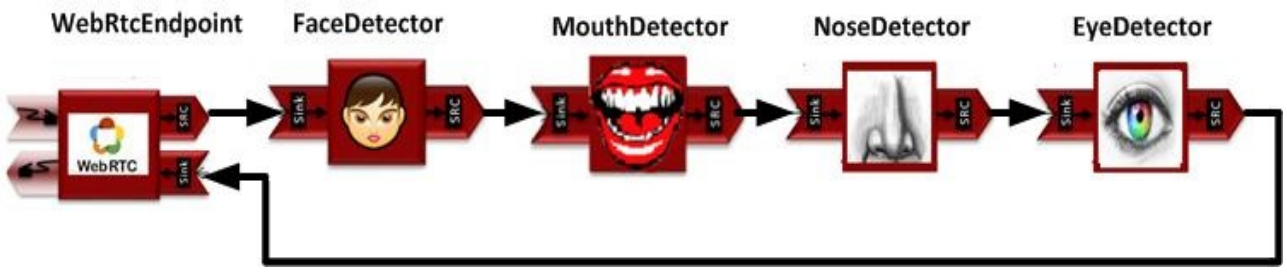

Figure 93: Face Mouth and Nose detector pipeline

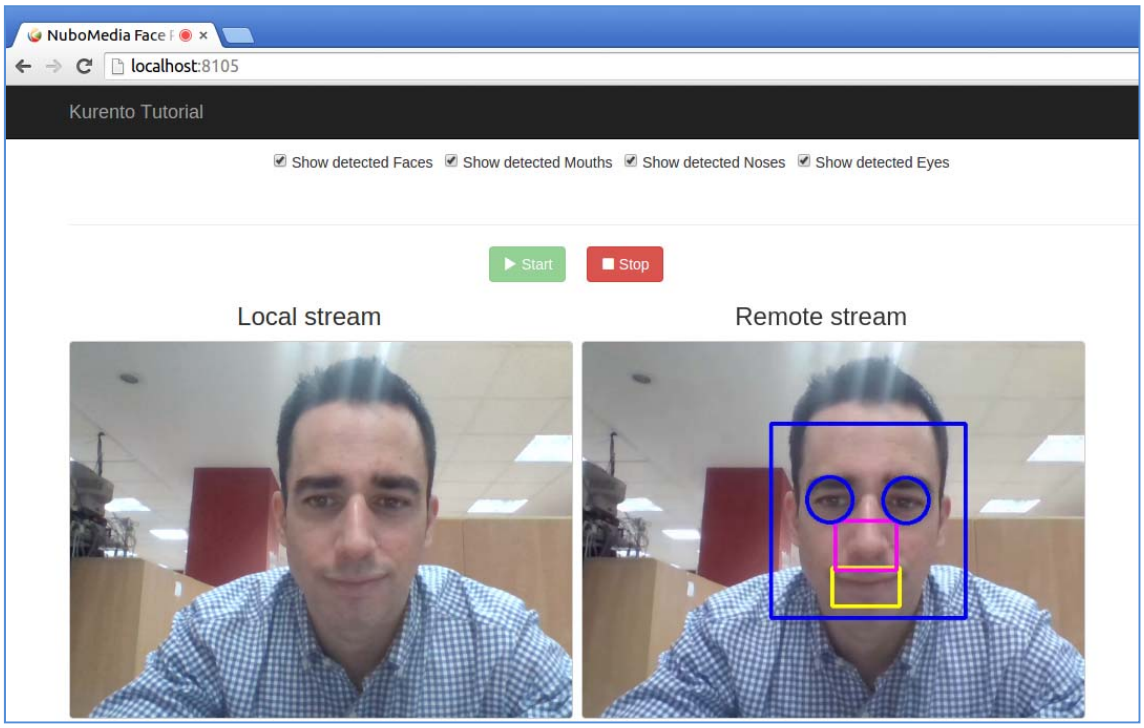

Figure 94: Face, Mouth and Nose detector running

Finally, it is important to highlight that we have created an on-line documentation for free and open source software (FOSS). Through this documentation you can see the installation and developers guide, the API documentation and the Architecture for the VCA modules of the NUBOMEDIA project. This documentation can be accessed on the following url:

- http://nubomedia-vca.readthedocs.org/en/latest/

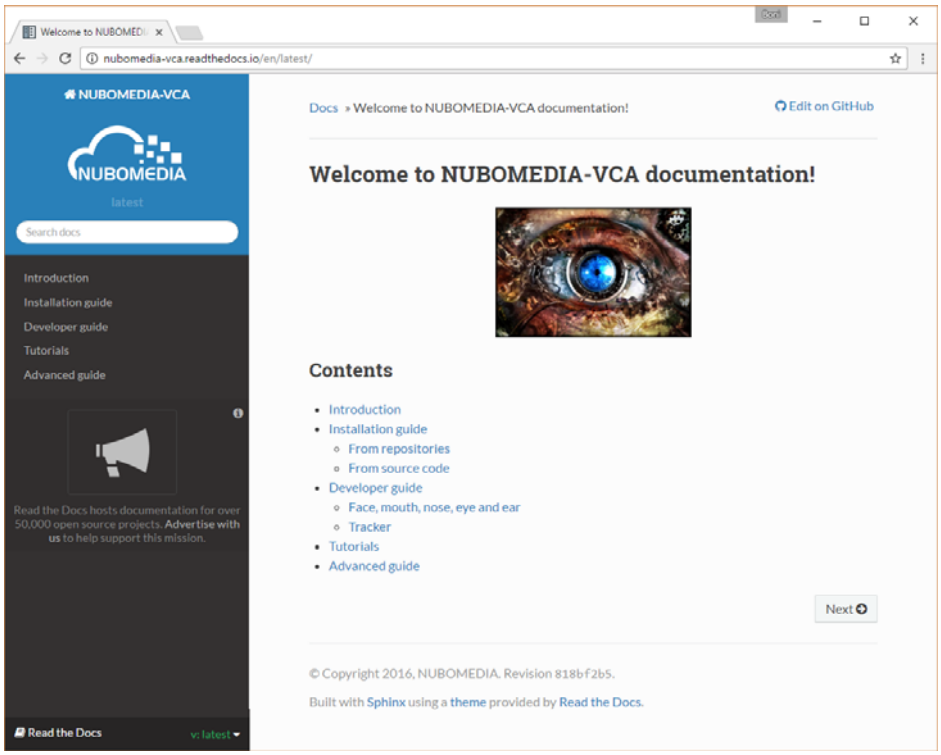

Figure 95: NUBOMEDIA-VCA documentation

In addition, the link to access the whole FOSS software can be found in the following url:

- https://github.com/VTOOLS-FOSS/NUBOMEDIA-VCA

The partners of the consortium can access to the source code of the proprietary software demos in the following url:

- http://80.96.122.50/victor.hidalgo/vca_media_elements.git

6 NUBOMEDIA Connectors

6.1 The NUBOMEDIA CDN Connector

6.1.1 Rationale

This section describes the NUBOMEDIA CDN Connector. The NUBOMEDIA CDN connector provides a data service API that can be used by applications to interface with an existing CDN networks.

A Content Delivery Network (CDN) which can also be called Content Distribution Network is a distributed infrastructure of proxy servers deployed in multiple data centers with the goal of serving end user content with high availability. A large amount of content (text, graphics, scripts, media files, software, documents, etc.) on the Internet is served by CDNs. There are three main types of CDNs, namely:

- *General purpose CDN*: performs web acceleration by using multiple servers on many locations, ideally close to large connection points between Internet Service Providers (ISPs) or also within the same data center (e.g. gaming data centers). Its main role is to cache and store content that is frequently requested by a high number of users
- *On Demand Video CDN*: performs the same role as a general purpose CDN but with the focus on just video content. These CDNs also provides a streaming server for video delivery. Streaming servers deliver the content at the time of a request, but only delivers the bits requested rather that the full length of the file. HTTP streaming with adaptive Bitrate (ABR) encoding and delivery is the technology used to efficiently deliver video clips.
- Live video CDN: despite ABR and HTTP streaming, there is demand for delivering live video content which cannot be cached. That is where live video CDNs come in. However, this category is the least mature of all 3 models.

For this project the focus is only on-demand streaming CDNs which are non – commercial. That is for which no fee is required. This is due to the following reasons:

- Majority (95%) of video content delivered by CDNs is on-demand video
- Live video cannot be cached. Thus, it will be required to modify the basic CDN infrastructure to have either very high-bandwidth pipes between central location and the end user viewing the content or to have slightly lower bandwidth pipes that send the live stream to a repeater or reflector that is near the end user
- The cost to maintain a live streaming solution for very poplar live events is expensive and most free CDNs offer this service with restrictions. For example, You-Tube, one of the most popular CDN offers the live-streaming functionality only to channels with more than 100 subscribers.

6.1.2 Objectives

The main objective of the CDN Connector is to evaluate current online user-generated content delivery providers who offer open APIs to access the user's data and provide a service to NUBOMEDIA users/applications to connect to one or more CDN via an API.

A number of use cases have been identified that will be interesting for NUBOMEDIA for the usage of such existing third party CDNs:

1) Upload a file to CDN
   Uploading a video file to the specific CDN which has been recorded before. This allows a user to upload their video call session to a CDN and publish this video content to users connecting to this CDN.
2) Discover videos stored on a CDN
   Provide an API that NUBOMEDIA clients can use to discover video content stored on the CDN.
3) Delete Videos
   Delete previously uploaded videos

6.1.3 Model Concept of CDN Connector

The CDN Connector is a Java-based library that delivers a unified service API to connect to registered CDN providers. It could also (on particular use cases such as upload video) connect with the NUBOMEDIA Media Plane to download a particular video file to upload on CDN.

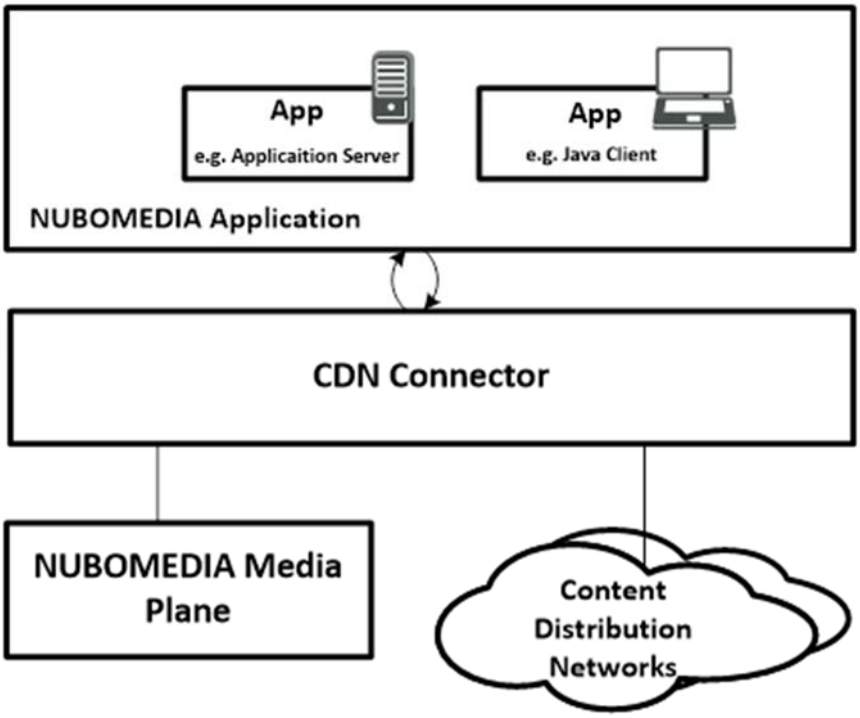

Figure 96 CDN Connector and Content Distribution Network

6.1.3.1 Applications

For this project, application model defines a NUBOMEDIA application as an application server residing on the server side, or as a Java client application residing on the client side. A NUBOMEDIA application can be implemented on some device,

perhaps using special hardware and software interfaces or Web based using Web standards.

6.1.3.2 CDN Connector

The CDN Connector is purposely designed to enable efficient management of connectivity to one or more CDN. There are two main functions of the CDN connector:

- Defines a CDN service API: a service interface that is exposed as an API for the above identified use cases
- Registering and unregistering of CDN providers: provider mechanism for supporting multiple CDN providers (e.g. YouTubeProvider, VimeoProvider, etc). Each registered provider has to implement the CDN service interfaces
- Routing messages to CDN providers: requests from the applications are internally routed to the message specific CDN provider

6.1.3.3 Media Plane

As already described extensively in this document, the NUBOMEDIA media plane is the NUBOMEDIA Media Server which provides a set of functionalities for creating media elements and media pipelines.

6.1.4 CDN Connector Service Architecture

This section explains in details the internal structure of the CDN Connector. As Figure 97 illustrates, the CDN Connector constitutes four main building blocks namely the CDN Manager, Session Manager, a collection of Connector Providers implementing the CDN Service API and the CDN SDK publicly provided by various CDNs for use by developers to access their distribution networks.

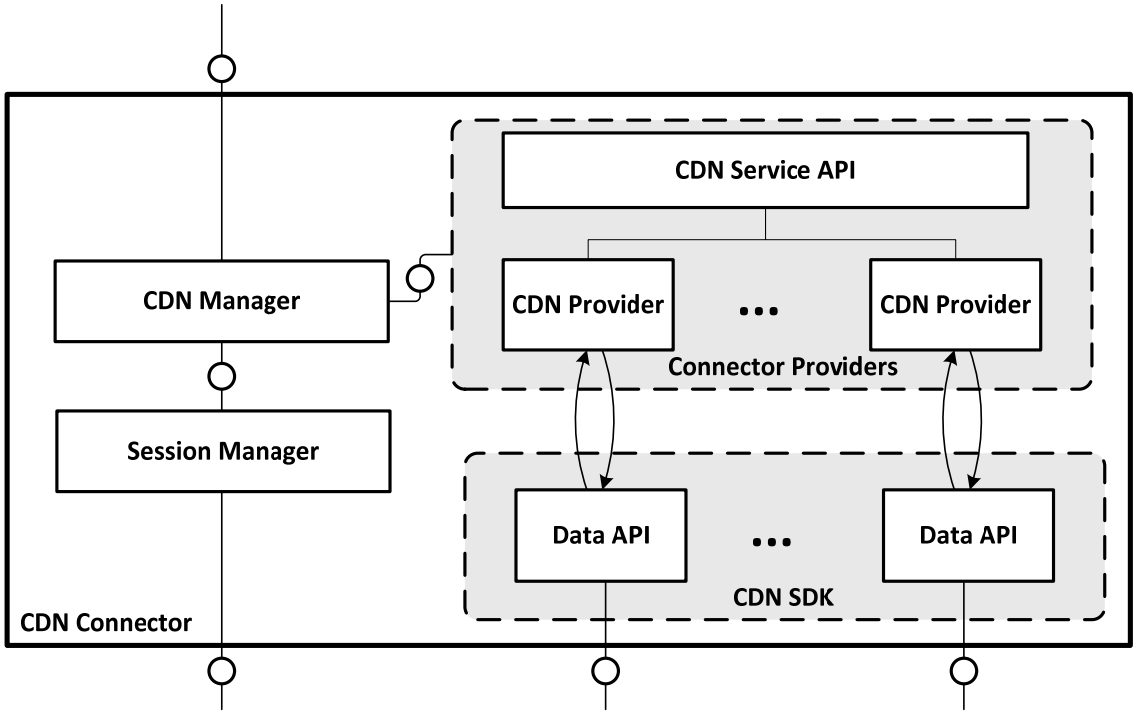

Figure 97 CDN Service Architecture

Let us look at each building block bottom up.

6.1.4.1 CDN Manager

The CDN Manager is the core module that manages all the other modules within the package. The functionalities of the Manager are:

- It holds collection maps for storing listeners, providers, dispatching, routing requests and triggering events.
- It holds an object of all registered providers. Developers wishing to provide their own implementation of a CDN Provider can do so and use the CDN Manager to register this provider.
- It holds an instance of the Session Manager for interactions with the NUBOMEDIA Media plane
- It provides a high abstraction service API of the CDN Service API to the NUBOMEDIA application.

Table 22 CDN Manager Service Interface

Service Interface	Description
`void registerCdnProvider(String scheme, CdnProvider provider, Json auth)`	Register a new CDN Provider. Input parameter is the scheme e.g. //youtube for a YouTubeProvider and a Json object with required credentials for authenticating to the registered user's data center.
`void unregisterCdnProvider(String scheme);`	Unregisters a CDN Provider from the collection of providers
`CdnProvider getCdnProvider(String scheme);`	Returns the specified CDN Provider from the collection
`SessionManager getSessionManager();`	Returns an instance of the Session Manager object
`uploadVideo(String sessionId) throws CdnException`	Uploads a new video file to the CDN. SessionId is the id of a previously stored video on the NUBOMEDIA cloud repository
`deleteVideo(String videoId, Json metadata) throws CdnException`	Deletes the video with the given identifier from the CDN
`getChannelList() throws CdnException`	Returns a JSON object with the list of all uploaded videos on the registered user's channel

6.1.4.2 Session Manager

The Session Manager provides the functionality to connect to NUBOMEDIA cloud repository where the Media Plane stores media files, download the files and make it available for uploading on the CDN. A use case where connection to the cloud repository is necessary is in the upload video use case. For example, a user might want to hold a conference session with one or two participants, and wishes to record the session on the Media Server and subsequently upload the recorded video to the CDN.

The address of the cloud repository is obtained from the implementation interface of the PaaS Repository Provider from the Media Client.

6.1.4.3 CDN Service API

The CDN Server API specifies an interface with service functions each CDN provider needs to implement. These functions provide an API to the NUBOMEDIA application

for accessing the CDN Connector services. As already introduced above, the use cases of interest are uploading, broadcasting, deleting and discovering video file stored in a registered user's data center.

Table 23 CDN Manager Service Interface

Service Interface	Description
`uploadVideo(String filename, Json metadata, Auth data) throws CdnException`	Uploads a new video file to the CDN. Metadata includes the title, description, tags of the video File name is the name of the downloaded file from the cloud repository. Authentication object with credentials needed to authenticate the user
`deleteVideo(String filename, Json metadata, Auth data) throws CdnException`	Deletes a previously uploaded video from the CDN. Input parameter is an Authentication object with credentials needed to authenticate the user
`getChannelList(Auth data) throws CdnException`	Returns a JSON object with the list of all uploaded videos on the registered user's channel. Input parameter is an Authentication object with credentials needed to authenticate the user

6.1.4.3.1 Upload Video File

NUBOMEDIA application that want to use this service need to send the initial session request to CDN Manager with the session identifier with which the file can be identified on the cloud repository. This request is dispatched to the Session Manager who is in charge of downloading this file from the cloud repository. With successful download, a notification is sent to `SessionManagerListener`, in this case the `CDNProvider` who then uploads the downloaded file on the CDN using the CDN's Data API SDK.

```
SessionManager.java:
void downloadFile(String fileName, SessionManagerListener handler){

        File folder = new File(downloadFolder);
                File download = new File(folder, fileName);
                try {
                        file.writeTo(download);
                } catch (IOException e) {
                ...
                }

                if(handler!=null){
                        handler.onFileDownloaded(download);
                }

}
. . .
```

There is also a listener interface `CDNProviderListener` which handles the results of the upload procedure accordingly. In this case, the `CDNManager` who forwards the response to the NUBOMEDIA application.

```
provider.uploadVideoAsFile(providerConfig, file, new ProviderListerner()
        {
                @Override
                public void onSuccess(String connectorId, JsonObject result)
                {
                  ...

                }

                @Override
                public void onError(ConnectorError e)
                {
                  ....
                }

        });
```

This interaction can be seen in the Figure 98 which shows the whole signaling flow between all components involved.

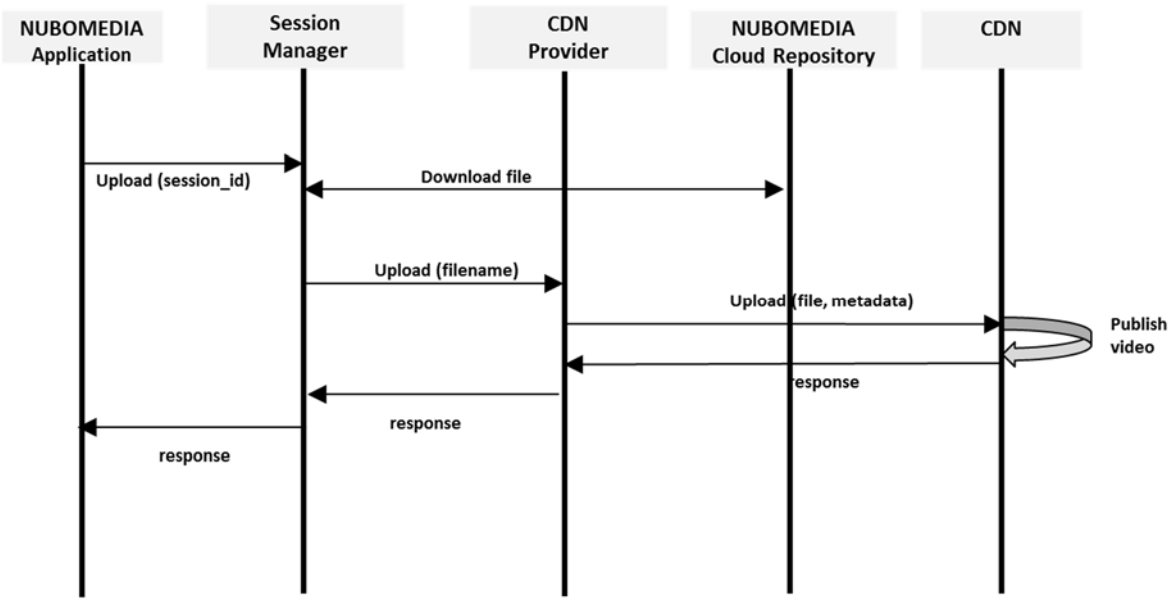

Figure 98: Download File from NUBOMEDIA Cloud Repository and upload file on CDN

6.1.4.3.2 Delete Video

The CDN Connector can also be used to delete a video file on the user's channel on the CDN. For this operation, the NUBOMEDIA application needs to send a request to the `CDNManager` who dispatches the request to the given `CDNProvider`. The following firgue shows the signaling flow:

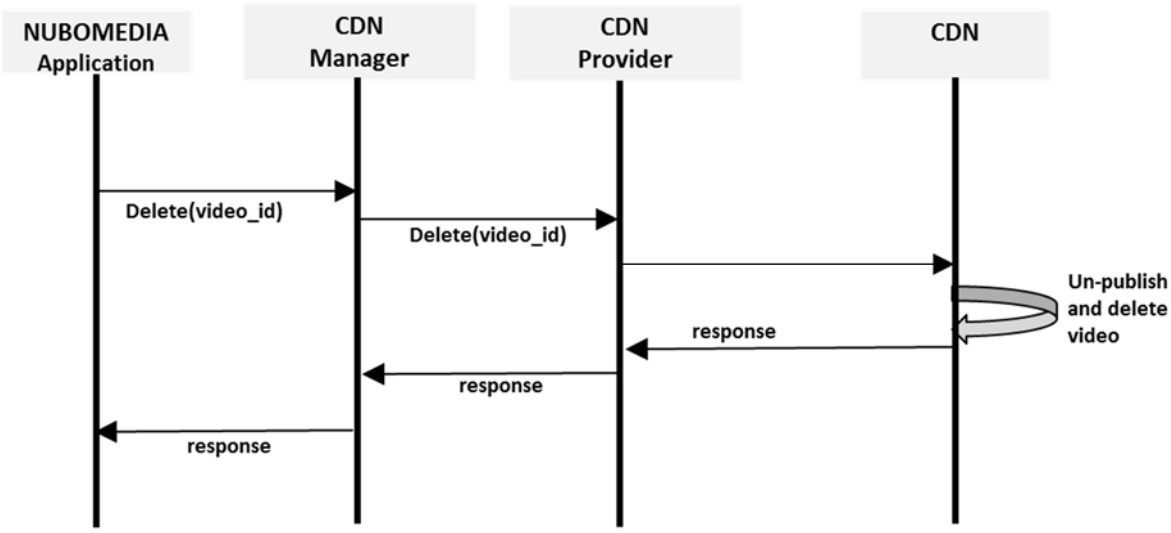

Figure 99: Delete video file on CDN

6.1.4.3.3 Get List of Videos on the User's Channel

The CDN Connector also provides the user the option to get the list of all uploaded video on his/her CDN channel. This information can further be used by NUBOMEDIA applications to query specific video content and metadata information, etc.

The NUBOMEDIA application send the request to the `CDNManager` and this is forwarded to the `CDNProvider` which uses its Data API SDK to connect and retrieve the list from the user's CDN. Examples of the video metadata are video identifier, upload date, tags and title.

The following picture shows the signaling flow:

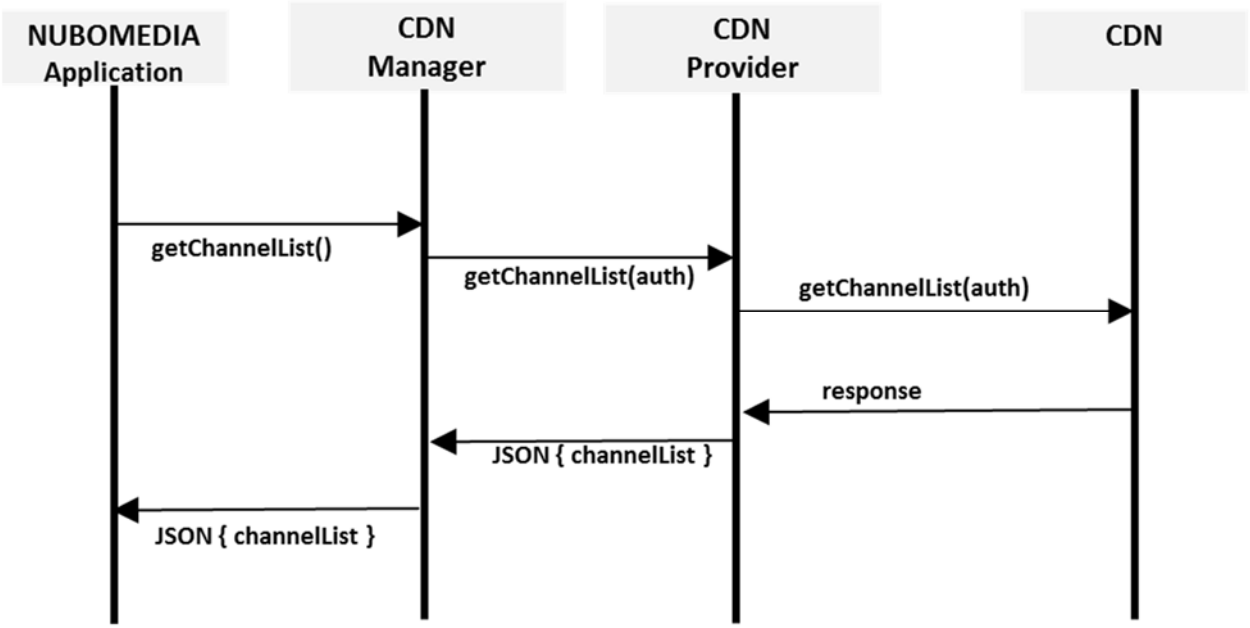

Figure 100: Retrieve all the uploaded videos from the user channel

6.1.5 CDN Software Development Kit (SDK)

The CDN Provider implements the CDN Service interface, providing functionalities to access and interact with the CDN's data center. Since each CDN provides a Data API for developers to use to access the data center, the CDN Provider provides an abstract layer on top of the Data API. The next sub sections briefly describe the most popular user generated CDN platforms and their Data APIs.

6.1.5.1.1 YouTube Provider

YouTube is the most popular user content generated CDN with site traffic of 52 Million monthly visitors. It provides a Data API that allows programmatic access to authenticated users to perform many of the operations available on the website. The API provides the capability to search for videos, retrieve standard feeds, and see related content. A program can authenticate as a user to upload, modify user playlists and broadcast a live stream.

6.1.5.1.2 DailyMotion Provider

Second to the list is DailyMotion with a monthly site traffic of 17Million.

6.1.5.1.3 Vimeo Data Provider

With a monthly site traffic of 17 Million visitors, Vimeo is the third most used user content generated CDN. Vimeo's Data API provides access to public data, share video functionalities with friends and change private settings to share videos with only selected people on the Vimeo network. The API uses RESTful calls and responses are formatted in JSON

6.1.5.1.4 Service and Data API comparison

Compare the functionalities

.

Table 24 General Information

Service	Owner	Launched	# videos	Views per day	main server	downloadable
Dailymotion	Dailymotion	March 15, 2005	>10,000,000	~60,000,000	France	No
Vimeo	Connected Ventures	November 2004	>1,000,000	1,000,000	United States	Not all videos allow download
YouTube	Google	February 15, 2005	>461,000,000	7,000,000,000	United States	Officially "Selected Videos Only".

Table 25 Supported input file formats

Service	MPEG	MOV	WMV	AVI	MKV	MP4	MOD	RA	RAM	ASF
Dailymotion	Yes	Yes	Yes	Yes	Yes	Yes	?	?	?	?
Vimeo[16]	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
YouTube[16]	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes

Table 26 Data API comparison

	You Tube	Vimeo	Meta Cafe	Daily Motion	Limelight	AWS
Video file upload API			?			
Upstream video API	Live streaming API + recording (Transcoding to RTMP needed)		?			
Downstream video API		Not directly, but something with download and file playback may be working	?		Not directly, but something with download and file playback may be working	Download URL
Playback video (over video player)	Proprietary player to be embedded into page	Proprietary player to be embedded into page	Proprietary player to be embedded into page	Proprietary player to be embedded into page		Via external Video player with downstream URL
Video file download API		If the uploader has activated the download feature (API?)	?		Download URI in the resource description	
Live broadcast video API	Transcoding of media stream to RTMP needed		?			

6.1.6 Evaluation and Validation

For evaluating and validating the CDN Connector, a YouTube CDN was implemented and registered on the CDN connector. The rest of this section describes how to configure and use the YouTube Provider.

The configuration of the CDN Connector module requires configuration parameter for the cloud repository that will be used for storing media files The generic structure of the configuration looks like this:

```
{
   "kurento_address": <the websocket URI of the kurento server that should be
used>,
   "repositroy_config": <configuration parameter of the repository to be used>
}
```

6.1.6.1 Repository Configuration

There might be different configuration parameters in the future according to the type of cloud repository that will be used. Currently only an implementation of a MongoDB FileFS repository is available. Therefore, the configuration may look like this:

```
{
    "repositroy_config": {
        "ip": <the IP address of the cloud repository>,
```

```
        "port": <the IP port where the coud repository is reachable> ,
        "user": <the user name>,
        "password": <the password of the user>,
        "database_name": (optional) <the name of the database that should be
used; default is "admin">,
        "collection_name": (optional) <the nameof the collection where the
files are stored; default value is "gridfs">,
        "downloadfolder":  (optional) <the folder where downloaded files will
be stored; default is "RepositoryDownloads">
    }
}
```

6.1.6.2 YouTube Provider

6.1.6.2.1 Prerequisites

When web client wants to make use of the YouTube Provider, some prerequisites must be fulfilled. This mainly relates to the creation of a YouTube account and a specific project associated with that account and the authorization to be able to make API requests.

1) **Create a YouTube account and a channel**
   registering an account for YouTube is quite easy: visit google.com and create a new user account. With the Google account a user can make use of the YouTube services. Create a channel for your account at https://www.youtube.com/channel (while logged in).

2) **Register a project**
   Visit https://console.developers.google.com (while you are logged in) and create a new project. On the left menu, go to "APIs & auth" -> "APIs" and enable the YouTube Data API

3) **OAuth configuration**
   YouTube uses OAuth 2.0 to authorize API requests. Therefore you need to configure credentials that will be used by your web application (aka NUBOMEDIA webrtc client) to retrieve an access token. The access token and other parameter will be sent with the client requests to the connector so that the YouTube connector is able to make YouTube API requests.

   In the left menu (same page like in step 2) go to "APIs & auth" -> "Credentials". Click on "Create new Client ID" -> "Web application". Then new credentials will be generated.

6.1.6.2.2 Request structure

As already explained, the requests of the CDN connector require a specific part for the CDN configuration:

```
    "cdn_config": {
        ...
        <specific configuration parameters of the CDN>
    }
```

These specific parameters are as follows for the YouTube Provider:

- Project name:
  The name of the project created in step 1 of the Prerequisites paragraph

- Access token:
  This token needs to be generated by the client using the OAuth credentials, the user account credentials and the Authorization server of Youtube.

- Client ID:
  The client ID which is part of the OAuth configuration parameters in part 3 of the Prerequisites paragraph

Summing up, the structure of specific configuration parameter for the Youtube connector must look like this:

```
"cdn_config": {
    ...
    "application_name": <this is the project name>,
    "auth":{
        "access_token": <the access token>,
        "client_id": <the cliend ID>
    }
}
```

6.1.6.2.3 Example Project

There is already an example of a web application for testing available at https://github.com/fhg-fokus-nubomedia/signaling-plane. This web application is based on the JavaScript examples provided by Google located at https://developers.google.com/youtube/v3/code_samples/javascript and implements the OAuth API. It shows how a web application can get an access token (and how it can upload video files to YouTube, but this part is not interesting for us).

The Web application of interest from the first URL is located in `src/main/resources/webapp`. In order to run a server with the webapp, have a look at the `src/test/org/openxsp/cdn/test/youtube/UploadTest.java` file. This class can be run as Unit Test which will start an HTTP server with this web application. Running the class as regular Java application will start a test which tests upload, delete and video discovery, but before the access token needs to be copied from the web console of the web application into the `src/test/org/openxsp/cdn/test/youtubeAuthTest.java` class (also client id and application name need to be adopted here to run the test).

6.1.6.2.4 Limitations

The YouTube connector only supports a subset of the CDN Connector services, this is partly due to the reason that video downloads are not supported by the YouTube API and that for live video broadcasting there is still some work for transcoding required. This means the YouTube connector currently supports only video upload, video discovery and deleting videos.

6.1.7 Implementation status

The initial version of the CDN Connector was provided as a module on the OpenXSP platform which is based on Vertx. The CDN Connector was used as a connection

between a NUBOMEDIA UA and CDN platform via the Event Bus of the Vertx platform.

This implementation, installation guide can be found at https://github.com/fhg-fokus-nubomedia/signaling-plane and developers guide can be found at https://github.com/fhg-fokus-nubomedia/signaling-plane/wiki

Due to complexity of the initial designed and deployed OpenXSP signaling plane and the complex and incomplete Platform-as-a-Service (PaaS) functionalities, it was decided within NUBOMEDIA WP3 to drop the OpenXSP as an Application Container.

Thus enhancements are needed on the CDN connector. The porting of the CDN Conector as a standalone library has already began and once stable will be available at https://github.com/fhg-fokus-nubomedia/cdn-connector. The final version of the CDN Connector can be found under the NUBOMEDIA GitHub repository, specifically at https://github.com/nubomedia/nubomedia-cdn-connector.

6.2 The NUBOMEDIA IMS Connector

6.2.1 Rationale

This section describes the NUBOMEDIA IMS Connector. The NUBOMEDIA IMS connector provides a 3GPP IP Multimedia Subsystem (IMS) service API that can be used by modules to interface with an existing IMS network.

The IMS is seen as an evolution of the traditional voice telephone systems, taking advantage of high speed data networking, multimedia, and increased processing power in user end points. However, many old principles are still preserved. An operator runs a network, allowing subscribers (e.g. Alice and Bob) to attach their end points to network endpoints. Alice and Bob get unique identities on the network. Alice can call Bob using his identity. The network creates a path between Alice's and Bob's devices for the call and they can start their exchange. The exchange part is where the most obvious difference is going from the single media of voice in telephony to multimedia in IMS

6.2.2 Objectives

The IMS Connector enables software developers to create feature-rich communications applications for Next Generation Networks and future mobile networks. The connector supports 3GPP IMS, GSMA RCS and Open Mobile Alliance (OMA) service enabler such as Presence, XML Document Management (XDM), SIMPLE IM-based instant messaging, and many other functions.

The key benefits and advantages provided by this module include:

- Shortens development time
- IMS stack builds on open standards from 3GPP (TS 24.229) and JSR 281 specifications

The key features include:

- Event Notification Framework for Presence and Location Information Publication, Subscription and Notification

- OMA Instant Messaging and Conferencing Multimedia Telephony with Audio and Video
- Chat with MSRP (Message Session Relay Protocol)
- Multimedia Telephony (3GPP MMTel)

6.2.3 Model Concept of IMS Connector

The IMS Connector is a Java-based framework that delivers a unified communication experience for all forms of 3GPP specified IP-based telecommunication. It is extremely powerful, yet lightweight enough to run on both client side and on server side. This connector provides developers with high level APIs for easy integration into their applications in order to enrich them with telco aware services. This service API will be referred to in the rest of this document as the IMS Service API (IMSAPI)

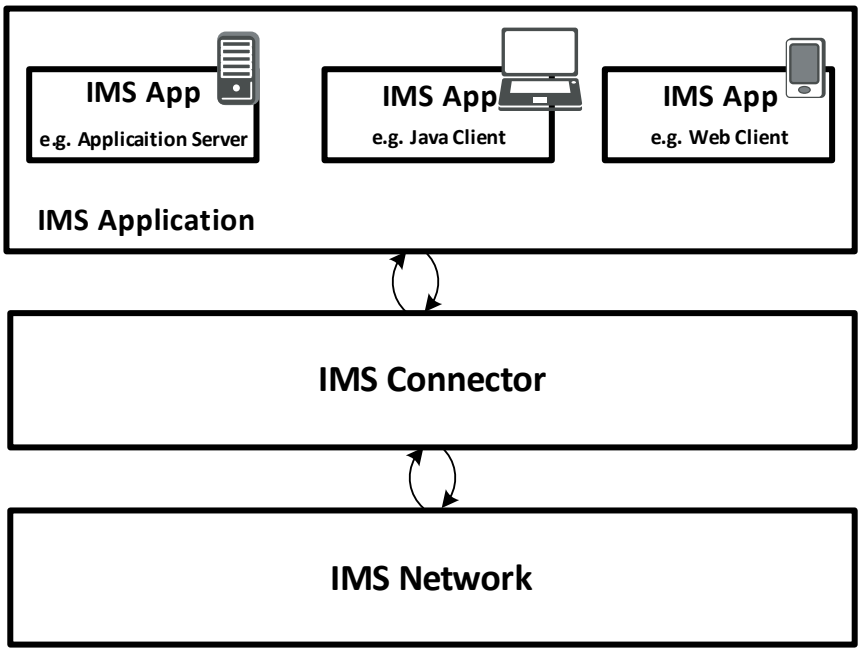

Figure 101 IMS Connector and IMS Application Model Concept

6.2.3.1 IMS Applications

For the NUBOMEDIA project, IMS application model defines an IMS application as a realization of one or more IMS capabilities for some well-defined end user purpose. This definition is useful because any arbitrary set of capabilities, or markers, does not make sense unless they are part of a function. It enables a simpler view of interoperability.

Note that the concept of an IMS application is independent of programming language, APIs, and devices. Even though the model is employed in IMS Connector, it is independent of its particular interfaces and Java.

An IMS application can be implemented on some device, perhaps using special hardware and software interfaces or Web based using Web standards. On the IMS network, any device with a particular IMS application implementation is a potential target for a successful IMS call. In this case, the network is responsible for routing an IMS call to the IMS Connector that subsequently routes the incoming call to the right IMS application.

6.2.3.2 IMS Applications in IMS Connector

The IMS Connector is purposely designed to enable efficient management of IMS applications in the NUBOMEDIA platform and interconnection to the IMS network. An IMS application realized within the framework of NUBOMEDIA, is called a NUBOMEDIA IMS application. There are two main aspects of an IMS application model:

- Registering and unregistering of IMS applications.
- Routing messages to IMS applications.

A NUBOMEDIA IMS application consists of the parent code whose execution defines the specific application service logic behavior, and a set of IMS properties that the IMSAPI implementation uses to manage correct operation towards the IMS network. Specifically, it plays a key role for interoperability. The parent code and the IMS properties are both necessary parts of an IMS application and are created at development time.

For this purpose, the IMS Connector provides a Registry which is a small database to register the capabilities of plug-in IMS applications. An IMS property set is maintained in the Registry, a small data base for the application.

6.2.3.3 IMS Application Identifiers

To be able to refer to a specific IMS application, an IMS application identifier (`AppId`) is used. The `AppId` is a symbol that follows the form of fully qualified Java class names or any naming syntax that provides a natural way to differentiate between application vendors. The `AppId` is set at development time, and MUST be unique within the context of the IMS application suite and the parent Java application suite.

6.2.4 Core Service Architecture

This section explains in details the internal structure of the IMS Connector. As Figure X illustrates, the IMS Connector constitutes three main building blocks namely the IMS API, Core Services, Registry and the Sip Stack.

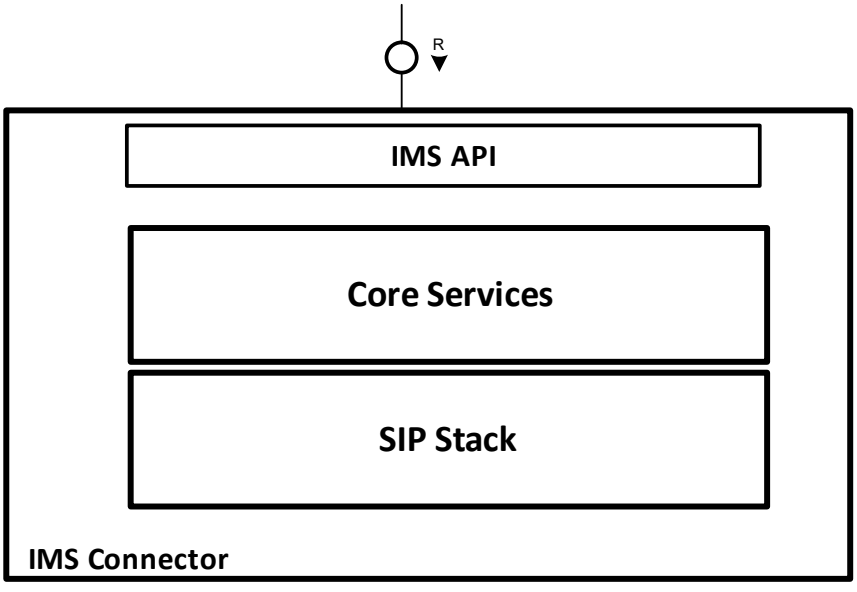

Figure 102 Core Service Architecture

Let us look at each building block bottom up.

6.2.4.1 Sip Stack

The Sip Stack used on the IMS connector is the JAIN-SIP reference implementation.
The Java APIs for Integrated Networks (JAIN) is a Java Community Process (JCP) work group managing telecommunication standards. JAIN SIP API is a standard and powerful SIP stack for telecommunications. The reference implementation is open source, very stable, and very widely used for SIP application development. The sip stack provides three base transport protocols, TCP, UDP and recently also WS.

6.2.4.2 Core Services

The core services package implements the JSR 281 specified interfaces that are used to register and configure IMS applications to the IMS Connector. The following functionalities are provided:

6.2.4.2.1 Connector

The parent application or IMS application connects to the IMS Connector using the `Connector.open` call where the connector argument string has the syntax: <scheme>:<target>[<params>]

- *scheme* is a supported IMS scheme. In this specification the imscore is defined in the `CoreService` interface
- *target* is an AppId. An AppId should follow the form of fully qualified Java class names or any naming syntax that provides a natural way to differentiate between modules.
- *params* are optional semi-colon separated parameters particular to the IMS scheme used

6.2.4.2.2 Service

Service is the base interface for IMS Core service. The service object is returned after an IMS applications calls the `Connector.open` to register to the IMS Connector. With the service object, the IMS application has access to the CoreService object, the AppId and scheme defined for the IMS application. With the CoreService object, the IMS application has access to the IMS API for creating IMS services.

6.2.4.2.3 Registry

The Registry consists of an unordered set of properties, which represent aspects of the IMS application. The properties include all capabilities, but there are also other types of information as well.
A property has a type name and zero, one, or more values. The number of values and their format depends completely on the type. A Registry contains unique properties, where the uniqueness depends on the property type.

Since an IMS Application must realize at least one IMS capability, it follows that a Registry must contain at least one of the IMS properties:

- *StreamMedia*: Declares that the application has the basic capability to stream audio and/or video media types
- *FramedMedia*: Declares a basic capability of messaging

- *BasicMedia*: Declares a basic capability to transfer media content of some MIME content type.
- *Event*: Declares a basic capability of handling event packages
- *CoreService*: Declare composed capabilities of core services

6.2.4.3 IMS API

The Core services are an implementation of the JSR281 specification. This services implement a collection of classes and interfaces that enable an application to create IMS services. Figure 3 depicts the static structure of the core services package.

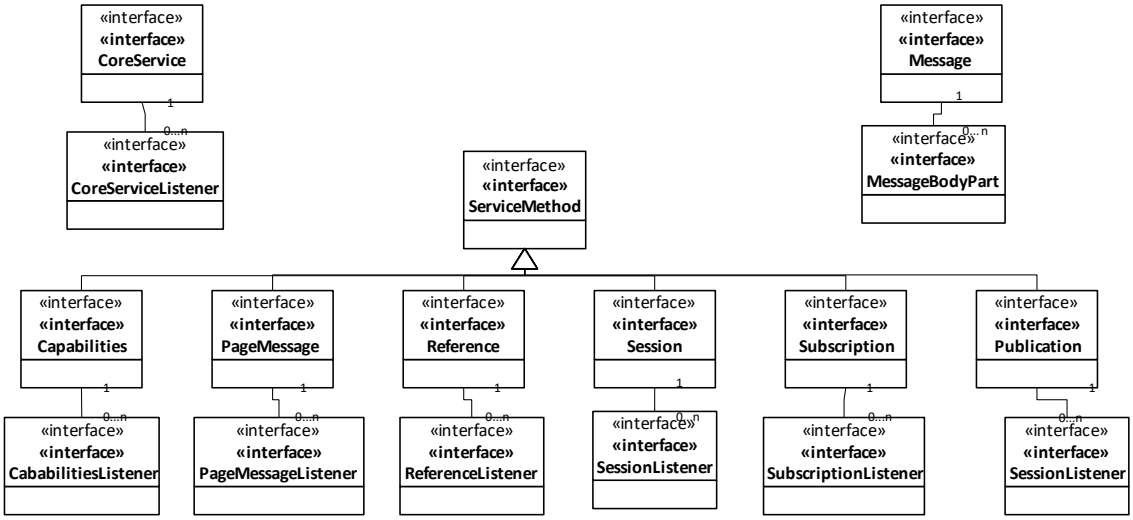

Figure 103 Core Services UML Package

The IMSAPI features a high level of abstraction, hiding the technology and protocol details, but can still prove to be useful and powerful enough for non-experts to create new SIP based IMS applications.

The SIPIMSAPI maintains the fundamental distinction between the signalling and the media plane in the API. It manages the SIP transactions and dialogs according to the standards, formats SIP messages, and creates proper SDP payloads for sessions and media streams.
SIP methods, with the transactions and dialogs they imply, have abstractions in the API. The core service object relates to IMS registration and the SIP REGISTER message.
The service methods correspond to non-REGISTER SIP messages and, where applicable, the session oriented SDP lines, sent to remote endpoints, including the standardized transactions and dialogs that the standards imply.

- Session - INVITE, UPDATE, BYE, CANCEL
- Reference - REFER and NOTIFY
- Subscription - SUBSCRIBE and NOTIFY
- Publication - PUBLISH and NOTIFY
- Capabilities - OPTIONS
- PageMessage - MESSAGE

The SIP INFO method has no representation in this API.

A parent application of IMS Application creates core service objects for an IMS application using a Connector.open as described in section 6.2.4.2.1.

It is recommended the IMS application registers itself as a `CoreServiceListener` in order to receive messages. The example below shows how to instantiate the core service object.

```
myCoreService = (CoreService)
Connector.open("imscore://org.nubomedia.CallApp");
myCoreService.setListener(this);
```

To stop execution, the module application closes each active service by calling its `CoreService.close` method.

With the `CoreService` instance, the IMS Application can create the following services:

- `CreateCapabilities`
- `CreatePageMessage`
- `CreatePublication`
- `CreateReference`
- `CreateSession`
- `CreateSubscription`

The `CoreService` Listener interface notifies the module on the following notifications

- `pageMessageRecieved`
- `referenceReceived`
- `sessionInvitationReceived`
- `UnsolicitatedNotifyReceived`

The subsections below briefly describes the different IMS services.

6.2.4.3.1 Page Message Service Interface

The `PageMessage` interface is used for simple instant messages or exchange of small amounts of content outside of a session.
The life cycle consists of three states, `STATE_UNSENT`, `STATE_SENT` and `STATE_RECEIVED`. A `PageMessage` created with the factory method `createPageMessage` in the `CoreService` interface will reside in `STATE_UNSENT`. When the message is sent with the send method the state will transit to `STATE_SENT` and further send requests cannot be made on the same `PageMessage`. An incoming `PageMessage` will always reside in `STATE_RECEIVED`.

Example of sending a simple `PageMessage`

```
try {
PageMessage pageMessage;
pageMessage = coreservice.createPageMessage(null,
"sip:alice@nubomedia.org");
pageMessage.setListener(this);
pageMessage.send("Hi, I'm Bob!".getBytes(),"text/plain");
} catch (Exception e){ //handle Exceptions }
```

Example of receiving a `PageMessage`
The callback method `pageMessageReceived` is found in the `CoreServiceListener` interface.

```
public void pageMessageReceived(CoreService service, PageMessage
pageMessage) {
pageMessage.getContent();
... // process content
}
```

6.2.4.3.2 Capabilities Service Interface

The `capabilities` interface is used to query a remote endpoint about its capabilities. An example looks like this:

```
public void queryCapabilities(CoreService service){
try {
Capabilities caps;
caps =service.createCapabilities(null,
"sip:alice@nubomedia.org");
caps.setListener(this);
caps.queryCapabilities(false);
}
catch(Exception e){
// handle Exceptions
}
}

public void capabilityQueryDelivered(Capabilities caps){
boolean feat =
caps.hasCapabilities("com.myCompany.myApp.video");
}
```

6.2.4.3.3 Publication Service Interface

The `Publication` interface is used for publishing event state to a remote endpoint. When a publication is created, an event package must be specified which identifies the type of content that is about to be published. A typical event package is `'presence'` that can be used to publish online information. Other endpoints can then subscribe to that event package and get callbacks when the subscribed user identity changes online status. The published event state will be refreshed periodically until `unpublish` is called. Updates of the published event state may be achieved by calling the `publish` method again with modified event state.

Example of a simple Publication

```
 try
{
pub =service.createPublication(null, "sip:alice@nubomedia.org",
"presence");
pub.setListener(this);
pub.publish(onlineStatus, "application/pidf+xml");
 } catch(Exception e){
 //handle Exceptions
 }

 public void publicationDelivered(Publication pub){
 //if the publish was successfull
 }
```

6.2.4.3.4 Reference Service Interface

The `Reference` is used for referring a remote endpoint to a third party user or service. The `Reference` can be created and received both inside and outside of a session. An

example scenario where a Reference can be used is to make a reference to a third party user. In this example, Alice tells Bob to establish a session with Charlotte.

```
try {
Reference ref =
service.createReference("sip:alice@nubomedia.org","sip:bob@
nubomedia.org ", "sip:charlotte@ nubomedia.org ", "INVITE");
ref.setListener(this) ref.refer();
}
catch(Exception e){ //handle Exceptions }
```

Bob receives the reference request and decides to accept it.

```
public void referenceReceived(CoreService service, Reference
reference) {
String referToUserId = reference.getReferToUserId();
 String referMethod = reference.getReferMethod();
// notify the application of the reference
...
reference.accept();
// assume referMethod == "INVITE"
mySession = service.createSession(null, referToUserId);
// connect the reference with the IMS engine
reference.connectReferMethod((ServiceMethod)mySession);
// start the reference with the third party
mySession.start();
}
```

6.2.4.3.5 Session Service Interface

The `Session` is a representation of a media exchange between two IMS Applications. A Session can be created either locally through calling `CoreService.createSession`, or by a remote user, in which case the session will be passed as a parameter to `CoreServiceListener.sessionInvitationReceived`.

The `Session` is able to carry media of the type `Media`. Media objects represent a media connection and not the media/content itself. The IMS media connections are negotiated through an offer/answer model. The first offer/answer negotiation may take place during session establishment. However, new offers may be sent by any of the session's parties at any time.

6.2.4.3.6 Subscription Service Interface

A `Subscription` is used for subscribing to event states from a remote endpoint. The subscription will be refreshed periodically until `unsubscribe` is called. To establish a subscription, a `Subscription` has to be created. The event package to subscribe to must be set, and the subscription is started by calling `subscribe`. If the remote endpoint accepts the request, a response will immediately be sent with the current event state that corresponds to the subscribed event package.

```
try {
 sub = service.createSubscription(null, "sip:bob@nubomedia.org",
"presence");
sub.setListener(this);
 sub.subscribe();
} catch(Exception e){
// handle Exceptions
}
```

```
 public void subscriptionStarted(Subscription sub){
  // if the subscription was successful
 }

 public void subscriptionNotify(Subscription sub, Message
notify){
 // check the subscribed event state
 }
```

6.2.5 Evaluation and Validation

This section evaluates what has been done so far in the usage of the IMS Connector. In general the IMS Connector provides the possibility to be used as an Application Server or as a User Agent. The IMS Connector itself does not need to be configured for a specific mode, it contains a configuration of the corresponding IMS node for communication. To use the NUBOMEDIA platform and the IMS Connector as application server in the IMS context, only an application server configuration in the HSS is necessary. The IMS Connector itself provides a grad variation in operation and can operate in both modes at the same time.

In general, the IMS Connector provides two modes of usage:

- as a User Agent
- as an application server integrated on the IMS network

6.2.5.1 IMS Connector as IMS User Agent

It is pretty straight forward to use the IMS Connector as an IMS User Agent within an application. As described above in section 6.2.4.2.1 and section 6.2.4.2.2, the application will need to connect and obtain an instance of the CoreService object in ordher to create IMS services, which can be integrated into the parent application.

6.2.5.2 IMS Connector as Application Server

An application server (AS) is a server which is brought into or notified of a session by the network in order to provide session features or other behavior.

In IMS, an application server is a SIP entity which is invoked by the S-CSCF for each dialog, both originating and terminating, as configured by the initial filter criteria (iFCs) of the relevant subscribers. The application server may reject or accept the call itself, redirect it, pass it back to the S-CSCF as a proxy, originate new calls, or perform more complex actions (such as forking) as a B2BUA. It may choose to remain in the signalling path, or drop out.

In the evaluation of the IMS Connector within the IMS network, it was used to redirect the call to the S-CSCF as a proxy.

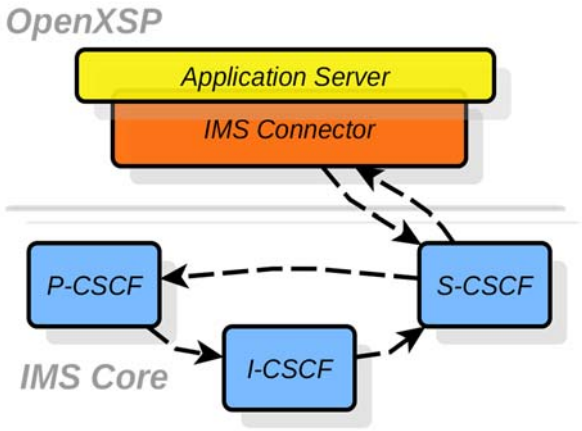

Figure 104 IMS Connector integration as Application Server

For the IMS network, the Open Source IMS Core (OSIMS) which was developed by Fraunhofer Institut FOKUS and now maintained by a spin-off company of Fraunhofer Institut FOKUS called Core Network Dynamics (CDN). OSIMS is an Open Source implementation of IMS Call Session Control Functions (CSCFs) and a lightweight Home Subscriber Server (HSS), which together form the core elements of all IMS/NGN architectures as specified today within 3GPP, 3GPP2, ETSI TISPAN and the PacketCable intiative. The four components are all based upon Open Source software (e.g. the SIP Express Router (SER) or MySQL). Installation of an instance of the OSIMS can be found at http://www.openimscore.org/documentation/installation-guide/.

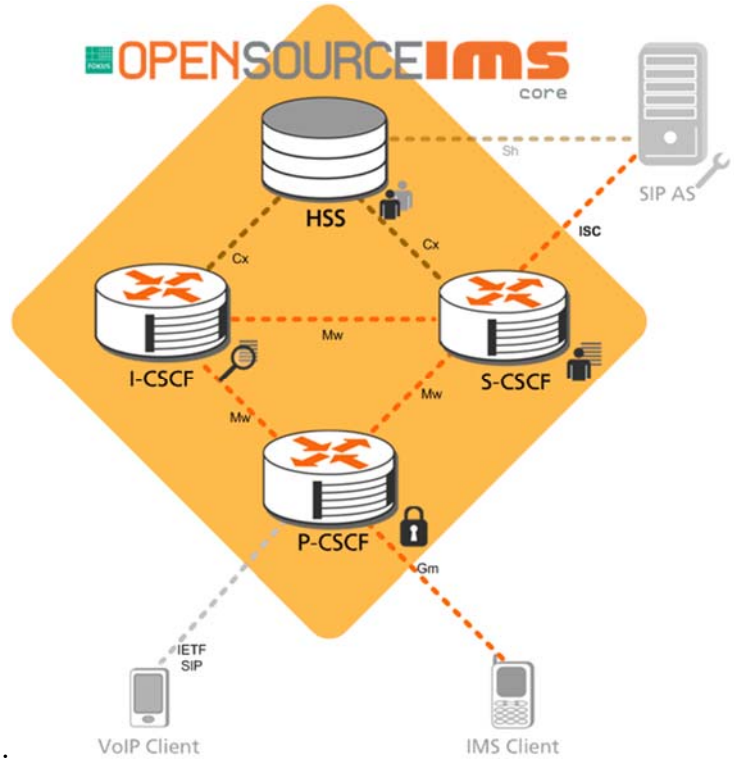

Figure 105 Open Source IMS Core

6.2.5.3 Adding a new Application Server

In order to add the IMS Connector as a new application server on OSIMS, the IP address in the form of SIP URI needs to be configured and the default behavior of the Serving-CSCF (S-CSCF) in case of connection failures.

All configurations are carried out through the FHoSS (FOKUS Home Subscriber Server) web interface. From FHoSS web console navigate to Services → Application Servers → Create and specify the following values:

- Name – any unique name,
- Server Name – actually it must be a valid SIP URI which resolves to application server address,
- Diameter FQDN – fully qualified domain name,
- Default Handling – default behaviour of the S-CSCF in case of connection errors,
- Service-Info – required, if used by the application.

6.2.5.3.1 Service Profile Creation

In order to create a service profile, you need to perform at most three steps:

- Create Trigger Point,
- Create Initial Filter Criteria,
- Create Service Profile itself.

The trigger point defines a set of conditions under which particular SIP request is forwarded to the application server. Particular conditions are provided by SPTs (Service Point Triggers) in the form of the regular expression. In order to create Trigger Points, navigate through: Services → Trigger Points → Create on the FHoSS web console and specify the following values:

- Name – any unique name
- Condition Type – logic by which SPTs are evaluated (conjunction, disjunction)

The Initial Filter Criteria defines a correlation between a set of Trigger Points and particular application server responsible for execution of the associated service logic. IMS architecture specifies that Initial Filter Criteria can have from 1 … n attached Trigger Points. In OSIMS it is simplified because there can be only one. So generally, in order to create Initial Filter Criteria the previously created Trigger Point needs to be specified together with the desired Application Server address. From FHoSS web console navigate to Services → Initial Filter Criteria → Create and specify the following values:

- Name – any unique name,
- Trigger Point – name of the already existing Trigger Point,
- Application Server – application server name,
- Profile Part Indicator – specifies the registration state condition for criteria evaluation.

The current used configuration is a trigger point rule on all SIP Invite messages that contains the user `rtccontrol` in the SIP TO header.

The final step involves assigning to the created service profile a prioritized list of Initial Filter Criteria including the newly created one. If this is not specified to then this Service Profile will not invoke and service logic and all requests on OSIMS will be processed without evaluation. From FHoSS web console navigate through: Services → Service Profiles → Create and specify the following values:

- Name – any unique name,
- IFCs – set of attached Initial Filter Criterias.

6.2.5.3.2 Creating User Account

The IMS user account is composed of three correlated parts:

- Subscription (IMSU) – identifies the user on the level of contract between subscriber and network,
- Private Identity (IMPI) – used by the user for authorization and authentication within the home network,
- Public User Identity (IMPU) – addressable identity of the user.

Any Service Profile is always prescribed to particular IMPU. This part describes how to create such account from the scratch. In order to create a user account, first all the above described steps need to be executed. To create IMSU from web console navigate through: User Identities → IMS Subscription → Create and specify the following values:

- Name – any reasonable unique name
- Capabilities Set – optional parameter specifying S-CSCF selection preferences for Interrogating-CSCF (I-CSCF),
- Preferred S-CSCF – optional preassigned S-CSCF.

To create IMPI navigate through: User Identities → Private Identity → Create and specify the following values:

- Identity – in the form username@domain i.e. alice@nubomedia-ims.test
- Secret key – password,
- Authentication Schemes – whichever is required, I usually use all,
- Default – for instance Digest-MD5.

Hereafter, the IMPI needs to be assigned to the previously created IMSU.
To create IMPU navigate through: User Identities → Public User Identity → Create and specify the following values:

- Identity – in the form of SIP URI i.e. sip:alice@nubomedia-ims.test
- Service Profile – any existing profile, there is always an empty profile created (no IFCs attached), which can be assigned by default.

At the end add a list of allowed visited networks, at least the home network and associate IMPU with previously created IMPI.

6.2.5.3.3 Attaching Service Profile to User Account

A Service Profile is activated for a particular user by assigning it to one of his IMPU's. From the HSS web console, navigate to: User Identities → Public User Identity → Search. Then set the Service Profile field with the appropriate value. Using the appropriate IMPU you can test and play with the services hosted on the plugged Application Server, in this case the IMS Connector.

6.2.5.3.4 Testing system with IMS and WebRTC clients

The reference architecture for WebRTC – IMS communication is described in 3GPP TR 23.701 and 3GPP TR 33.871. It is unrealistic within this project to achieve a reference implementation to this architecture as many components need to be developed and tested with the IMS network. However, we have identified the basic building blocks to simulate a proof of concept towards the referenced architecture. Figure 6 illustrates this proof of concept test-bed. The following test cases have been successfully tested. It should be noted, that in the case of interoperability between the IMS and WebRTC

client, ONLY signaling has been successful. In other to include the media path, a WebRTC Media Gateway is needed which is beyond the scope of this task.

```
REGISTER ----------> P-CSCF
     401 <---------- P-CSCF
REGISTER ----------> P-CSCF
     200 <---------- P-CSCF


  INVITE ----------> P-CSCF
     100 <---------- P-CSCF
     180 <---------- P-CSCF
     200 <---------- P-CSCF
     ACK ----------> P-CSCF


REGISTER ----------> P-CSCF (DeREGISTER Message)
     401 <---------- P-CSCF
REGISTER ----------> P-CSCF
     200 <---------- P-CSCF
     BYE <---------- P-CSCF
   Pause [    20.0s]

     ACK ----------> P-CSCF

     200 <---------- P-CSCF
```

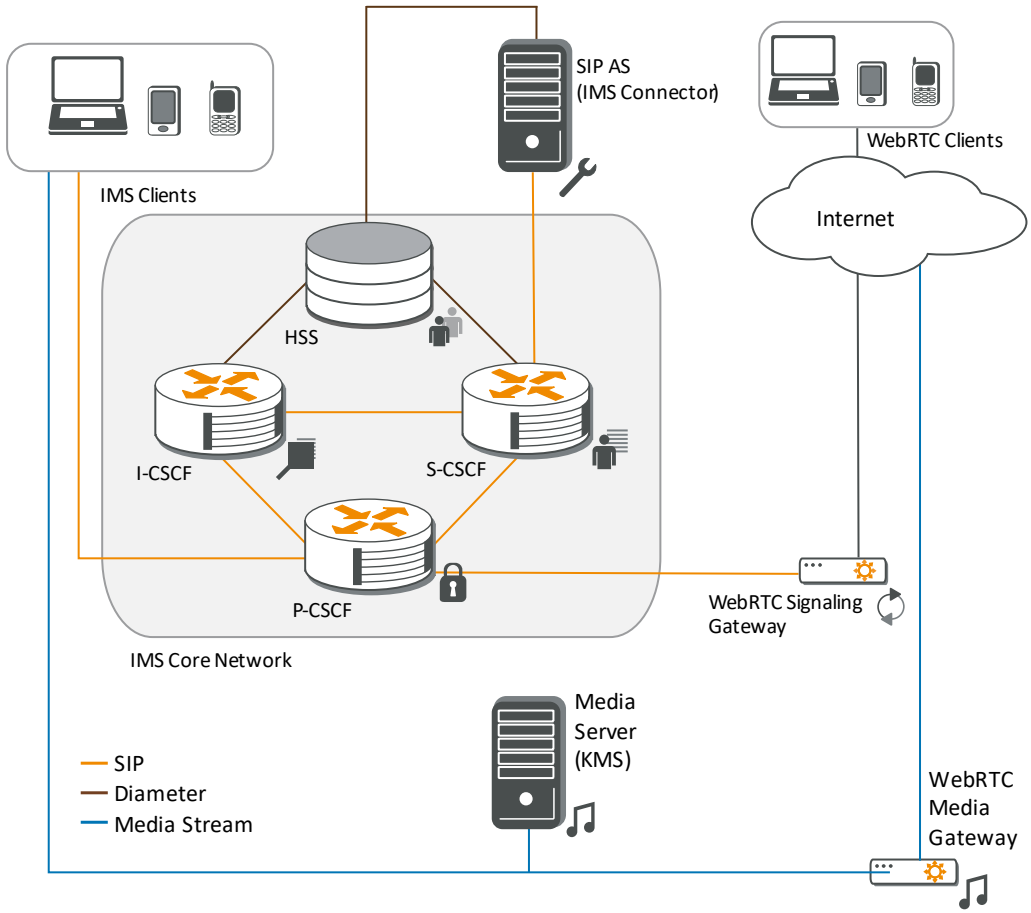

Figure 106 NUBOMEDIA IMS Testbed

6.2.6 Implementation status

The initial version of the IMS Connector was provided as a module on the OpenXSP platform which is based on Vertx[]. The IMS Connector was used as a connection between an IMS core network and the Vertx platform. From the IMS core network

perspective, the Connector is acting as an Application Server. The address and trigger rules were part of the IMS core network, and were configured in the corresponding HSS. The trigger point rule gets executed in the S-CSCF, the communication is restricted to the S-CSCF and the IMS Connector. The current used configuration is a trigger point rule on all SIP Invite messages that contains the user `rtccontrol` in the TO header. The IMS Connector is treated as application server, and acts like a UA in IMS environment outside of the IMS core.

This implementation, installation guide can be found at https://github.com/fhg-fokus-nubomedia/signaling-plane and developers guide can be found at https://github.com/fhg-fokus-nubomedia/signaling-plane/wiki

Due to complexity of the initial designed and deployed OpenXSP signaling plane and the complex and incomplete Platform-as-a-Service (PaaS) functionalities, it was decided within NUBOMEDIA WP3 to drop the OpenXSP as an Application Container.

Thus enhancements are needed on the IMS connector. To be specific, the IMS Connector needs to be ported out the OpenXSP platform and provided as a standalone server.

The final version of the IMS Connector can be found under the NUBOMEDIA GitHub repository, specifically at https://github.com/nubomedia/nubomedia-ims-connector.

6.2.6.1 Extension as a SIP Servlet on Mobicents Jain-Slee

Mobicents is the leading Open Source VoIP Platform. It is the First and Only Open Source Certified implementation of JSLEE 1.1 (JSR 240), and SIP Servlets 1.1 (JSR 289)

6.2.6.2 Integration with Kurento Media Server (KMS)

So far, the focus has been concentrated on providing interconnectivity to the IMS network from the OpenXSP platform, and later interconnectivity to WebRTC clients. Signaling wise, test have been successful but media path integration has been rather more of a challenge. This said, a possible solution would be to place the KMS on the media path such that media from two IMS UA can be managed from the KMS.
For this to happen, the IMS Connector needs to be extended with two functionalities:

- Implementation of a B2BUA
- Implementation of the service logic to include the Kurento Media Client API for communicating with the KMS

6.3 The NUBOMEDIA TV Broadcasting Connector

6.3.1 Rationale

The TV Broadcaster connector as described in the DoW is devoted to conceiving and creating the appropriate media pipeline for the generation of live feeds, near live and non-live content for TV broadcaster's services. Complying with NUBOMEDIA platform distributed media elements, this connector, will allow developers to incorporate into their applications a direct and live connectivity with broadcasters. In order to develop the appropriate interface, we have first gathered the requirements and

this document is providing the high-level specification based on that and the architecture developed in WP2.

6.3.2 Objectives

The TV Broadcasting Connector main objectives is to create a media path bridging between content exchange platforms/ social TV or connected TV applications and the TV broadcasters. Thus, the TV Broadcasting Connector can connect between:

- Users generating content with content publishers,
- Publishers to Publishers,
- Publishers to users,
- Users to user's

The objective of the connector is to hide the complexity of the service media path by providing an easy interface to connect among these various mentioned stakeholders.

The TV Broadcasting Connector will connect streams coming from such services to a dedicated delivery system that will handle the media distribution in an effective manner to the broadcaster's studio. The impact could bring novel social and connected TV services to be used by many users.

6.3.3 Model Concept of TV Broadcasting Connector

For the definition of the different involved interfaces, LIVEU have considered real TV broadcaster scenario where broadcasters can benefit from. The connector will make the connection between users and broadcaster applications through a dedicated third party service, for testing that we are developing it within WP6 demonstration a content exchange platform what we call NewsRooms, this platform will bridge between TV broadcasters, and TV media distribution networks; thus, media applications developers and users/consumers could easily develop connected TV applications and share content with the broadcaster's platforms.

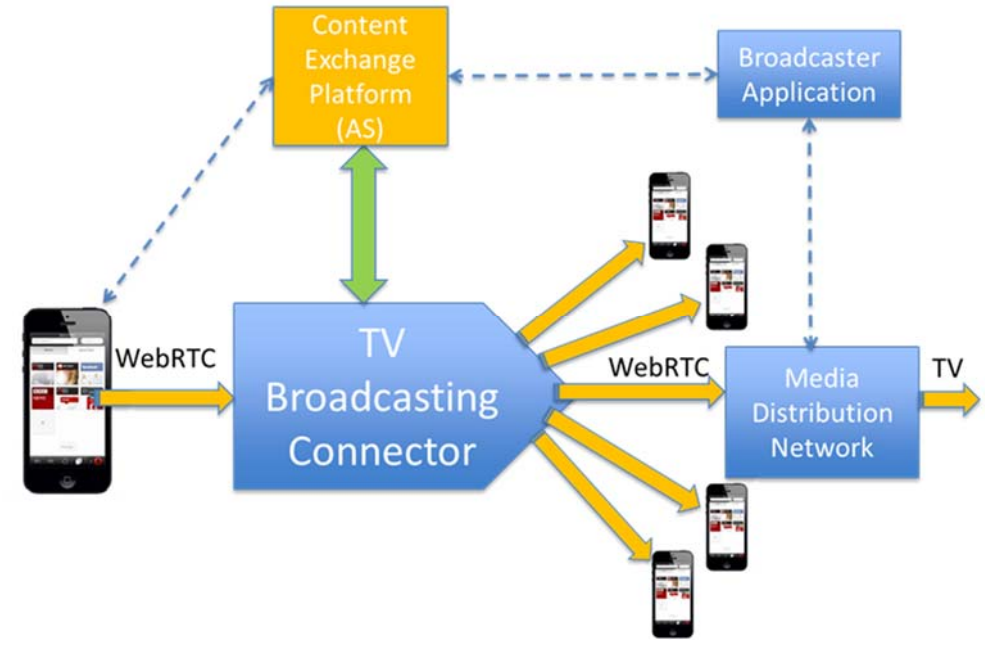

Figure 107: TV Broadcasting Connector and the Environment

6.3.3.1 Content Exchange Platform

LiveU develop technologies to allow any user using their mobile browser and WebRTC streaming to send video to broadcasters through LiveU cloud services. The content exchange platform is an application server aiming to connect between consumers and broadcasters (Main use case as described in D2.1.2) "NewsRoom". this scenario goal is to connect the broadcasters with the consumers in order to increase further the richness and freshness of the content, when an unplanned event with public interest occur, consumers are the first to be there.

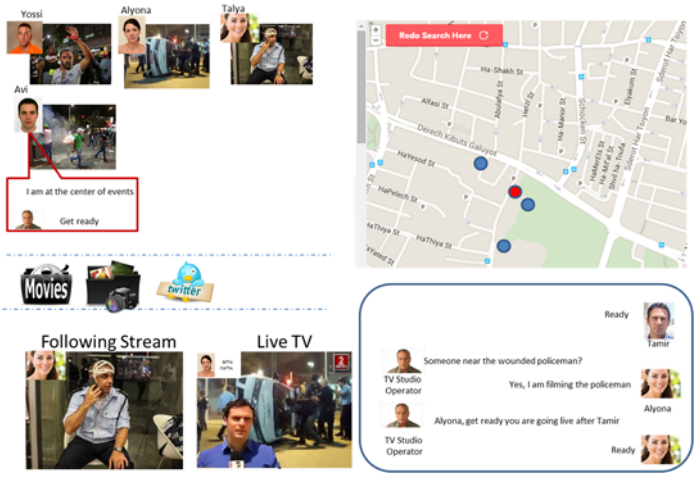

Figure 108: Content Exchange platform: NewsRooms

6.3.3.2 Media Distribution Network

The media distribution network consists of high definition and reliable cloud streaming servers which is connected through a set of MMH units (LU2000) organized in a tree distribution from.

Figure 109: Media Distribution Networks and MMH

LiveU's Distribution Solutions provide solutions for sharing high quality live video between multiple broadcast facilities over the public Internet. LiveU MultiPoint seamlessly integrates into users daily workflow, allowing them to share incoming live feeds with multiple end-points, from within the same LiveU Central interface they work with daily.

- Large-scale distribution - up to 100 concurrent destinations
- Cost effective solution over public Internet for distribution of Live media to any destination
- No additional Hardware required - Based on existing LU2000 server
- Integral part of LiveU Central – can be controlled from anywhere
- Adaptive video streaming using LRT (LiveU Reliable Transport)
- Pay per usage business model for maximum flexibility
- Low Delay – 0.5 seconds per hop

6.3.3.3 Broadcaster's Application & Servers

Unified management platform for easy preview and remote control. The LiveU Central management platform allows Broadcasters the full control and monitoring of the entire ecosystem and content via any browser-supported computer or tablet, from anywhere around the world.

Figure 110: Broadcasters Applications - LU Central

6.3.4 Core Service Architecture

The TV Broadcasting Connector allows consumers to send video directly from their mobile browser to TV broadcasters (the connector is handling the related control and media path for the video end to end through an API). The Architecture of the broadcasters TV Connector is described in Figure 111 below. The IPTV Connector takes part of the Live Mobile and TV Broadcasters Scenario defined in D2.1.2.

The orange elements of the architecture represents environment elements that takes part in the demonstration, these elements are either exist or modified to support the demonstrator in WP6 activities. The blue elements are the TV Broadcasting Connector WP4 elements which consists of the AS – a new module that will be responsible on communicating with the AppServer (content exchange server) to establish the required Media Pipelines as part of the media path.

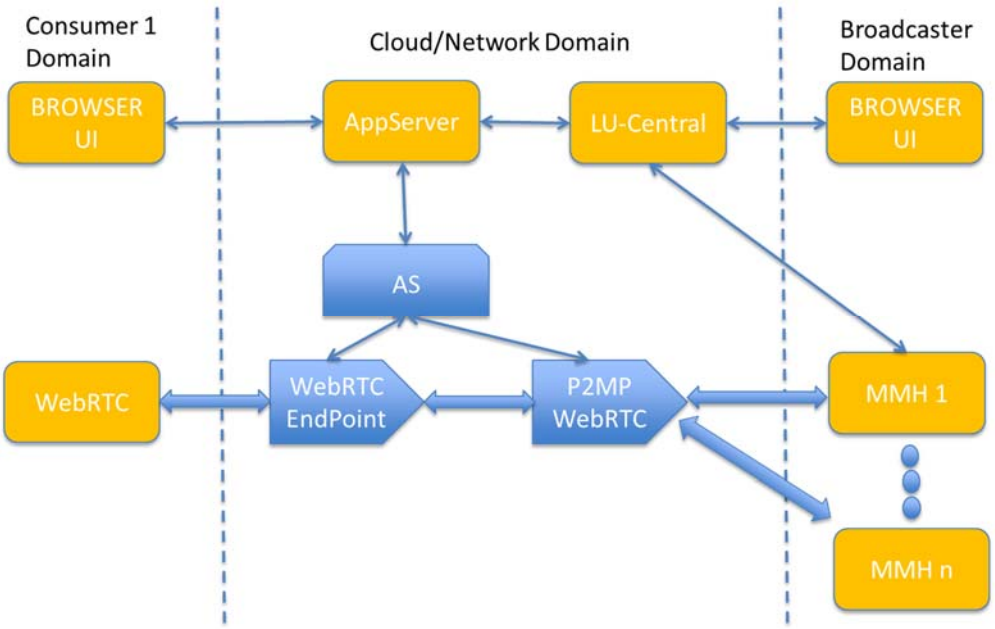

Figure 111: TV Broadcasting Connector Architecture

In Order to stream video to the Broadcasters, the AppServer shall provide all the information to the consumer (i.e. where to stream the WebRTC stream and other proprietary data related to the AppServer service). In Parallel, the AppServer shall establish and manage the pipeline through dedicated TV Broadcaster API. With the provisioned configuration set by the broadcaster at the LU-CENTRAL (Broadcasters Application) the Multi-Media Hub (MMH) which is the media distribution network end point will be receiving the stream in WebRTC protocol.

6.3.4.1 Establishing a Media Pipe for the consumer

Once the consumer is streaming his live video the WebRTC endpoint will receive the stream, will pass it to the WebRTC Point to Multipoint (P2MP) filter to stream the media via WebRTC protocol to the different MMH's provisioned by the broadcasters. This can be seen in the following sequence flows.

In this sequence, we can see how the example of how consumer connects to the service.

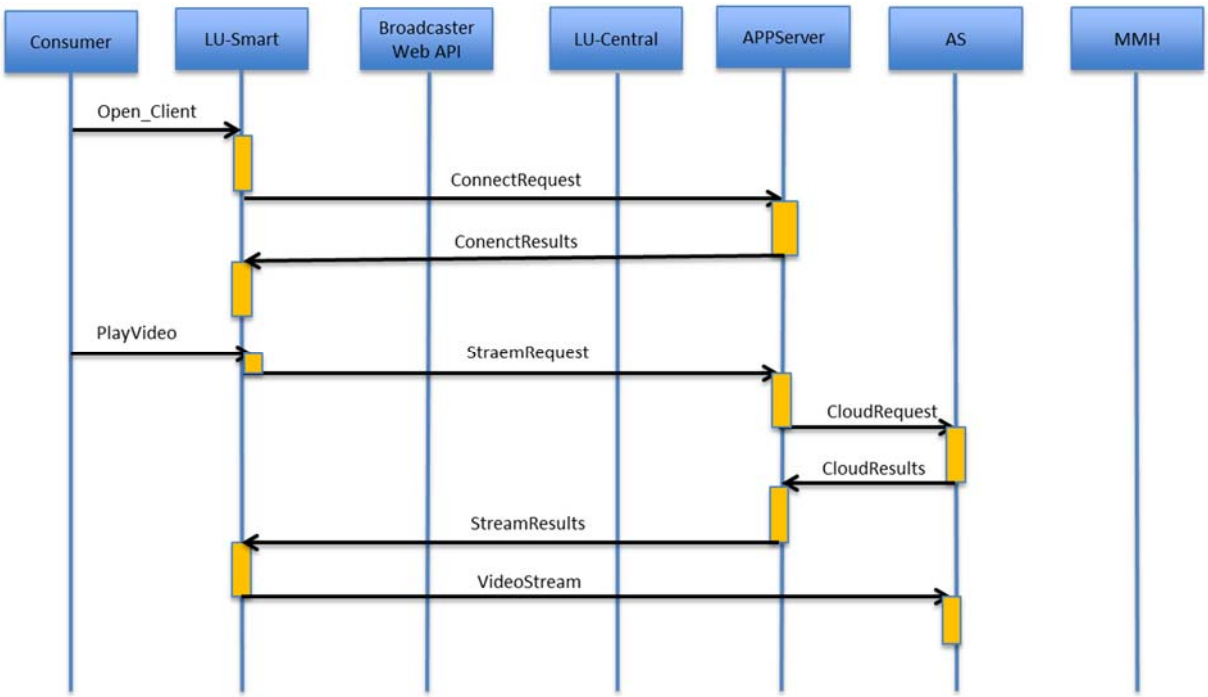

Figure 112: Sequence diagram for LU-Smart Connection in Live TV broadcasting use case.

Then, the different primitives corresponding to this diagram are explained:

- Open_Client this primitive is for the Consumer to open the app/browser pointed to the WebServer
- StreamRequest this primitive the LU_Smart request to connect to the web service
- ConnectResults in this primitive the WebServer respond to the ConnectRequest function
- PlayVideo in this primitive the consumer is pressing the play button to start the video stream.
- StreamRequest in this primitive the LUSmart is requesting to stream from the WebServer.
- Cloudrequest in this primitive the AppServer is requesting from the AS to create a pipeline for the LUSmart stream
- CloudResult in this primitive the AS reply to the AppServer with the pipeline results
- StreamResuts in this primitive the WebServer is replying to the LUSmart with the StreamRequest results.
- VideoStream in this primitive the LUSmart is streaming the video to the AS

6.3.4.2 Streaming the video to the broadcaster

The Consumer holding can start live streaming after successful connection. The consumer can change some of the streaming parameters (e.g. video resolution). The stream will be sent to the TV Broadcasting connector, where it may be stored or also

forward to MMH-server and from there to the broadcaster through CDN or direct SDI interface depending on the MMH provisioning.

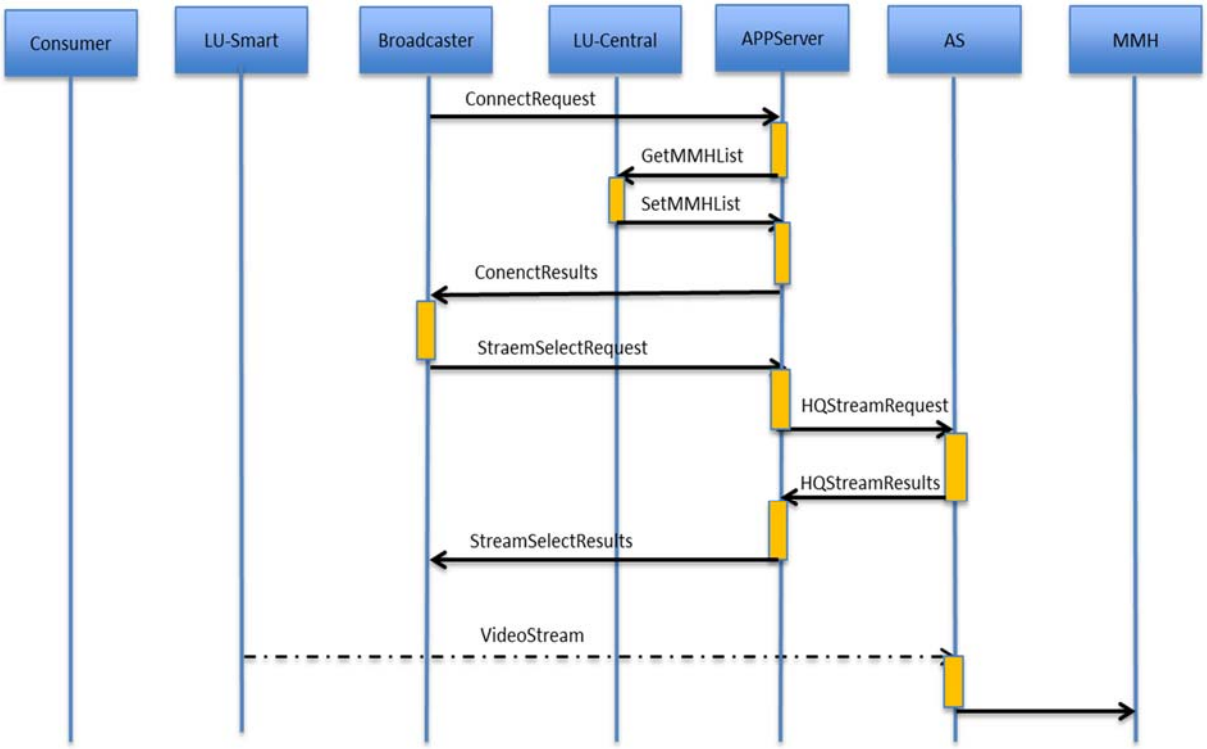

Figure 113: Sequence diagram for streaming to the broadcasters

Then, the different primitives corresponding to this diagram are explained:

- ConnectRequest in this primitive the broadcaster is connected to the AppServer
- GetMMHList the App server is requesting from the LUCentral the list of available MMH servers
- SetMMHList the LU Central is providing a response with the list of available MMH to the AppServer
- ConnectResults the AppServer is providing a response to the ConnectRequest of the broadcaster
- StreamSelectRequest the Broadcaster is selecting a stream to be broadcasted in his network
- HQStreamRequest the AppServer is requesting a HQ stream from the AS
- HQStreamResponse the AS response to the HQStreamRequest
- StreamSelectResponse the AppServer is replying to the broadcaster about the StreamSelectRequest resonse
- The AS is forwarding the stream of the LUSmart to the selected MMH

6.3.5 Detailed API

6.3.6 Detailed API

TV Connector API is organized into Java classes in package tv.liveu.tvconnector.
Relations between API classes and main platform components are shown at Figure 9.
API classes has blue color.
Package and classes documentation are described below.

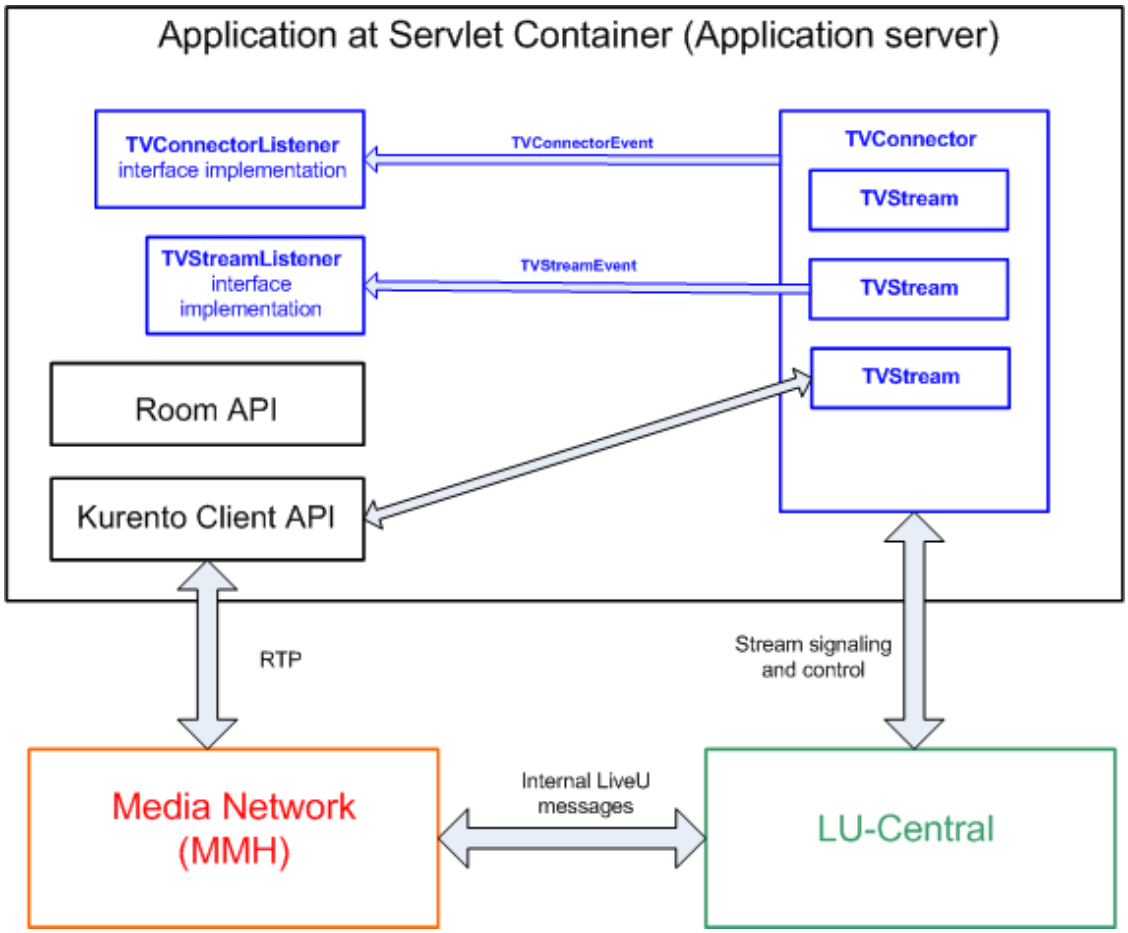

Figure 9: TV Connector classes diagram

6.3.6.1 *Package tv.liveu.tvconnector*

Interface	Description
TVConnectorListener	This interface is for classes that wish to receive TV Connector events.
TVStreamListener	This interface is for classes that wish to receive TV Stream events.

Class	Description
TVConnector	Connection to TV provider.
TVConnectorEvent	This event is generated for a TVConnector event.
TVStream	Video, audio TV stream.
TVStreamEvent	This event is generated for a TV stream event.

6.3.6.2 Class TVConnector

Connection to TV provider. Handle media, control streams and TV broadcast related functionality.

Constructor and Description

`TVConnector()`

- **Method Summary**

All Methods

Modifier and Type	Method and Description
void	close()
TVStream	getTVStream(java.lang.String name)
java.util.ArrayList<java.lang.String>	getTVStreamList()
void	onStreamPublised()
void	open(java.lang.String url)
void	publishTVStream(TVStream stream)

Constructor Detail

- **TVConnector**

```
public TVConnector()
```

Method Detail

- **close**

```
public void close()
```

- **getTVStream**

```
public TVStream getTVStream(java.lang.String name)
```

- **getTVStreamList**

```
public java.util.ArrayList<java.lang.String> getTVStreamList()
```

- **onStreamPublised**

```
public void onStreamPublised()
```

- **open**

```
public void open(java.lang.String url)
```

- **publishTVStream**

```
public void publishTVStream(TVStream stream)
```

6.3.6.3 Class TVConnectorEvent

This event is generated for a TVConnector event. Event can be kind of:

- open-close connection to Central
- create-destroy `TVStream`
- stream control and authentication events

Fields

Modifier and Type	Field and Description
`static int`	CONNECT_FAILED This event indicates that connection is failed
`static int`	CONNECT_SUCCESS This event indicates that connection is successfully opened
`static int`	STREAM_CREATED This event indicates that new stream created
`static int`	STREAM_DESTROYED This event indicates that stream is destroyed

Constructor and Description

TVConnectorEvent`(int id, java.lang.String message)`
Initializes a new instance of `TVStreamEvent` with the specified information.

- **Field Detail**
  - **CONNECT_FAILED**

```
public static final int CONNECT_FAILED
```

This event indicates that connection is failed

  - **CONNECT_SUCCESS**

```
public static final int CONNECT_SUCCESS
```

This event indicates that connection is successfully opened

  - **STREAM_CREATED**

```
public static final int STREAM_CREATED
```

This event indicates that new stream created

  - **STREAM_DESTROYED**

```
public static final int STREAM_DESTROYED
```

This event indicates that stream is destroyed

- **Constructor Detail**

- **TVConnectorEvent**
- `public TVConnectorEvent(int id,java.lang.String message)`

Initializes a new instance of `TVStreamEvent` with the specified information. Invalid id leads to unspecified results.

Parameters:
`id` - the event id
`message` - event description

6.3.6.4 *Interface TVConnectorListener*

This interface is for classes that wish to receive TV Connector events.

- **Method Summary**

Modifier and Type	Method and Description
`void`	`connectDone(TVConnectorEvent event)` This method is called when TVConnector connect attempt is done
`void`	`connectionDestroyed(TVConnectorEvent event)` This method is called when TVConnector connection destroyed
`void`	`streamCreated(TVConnectorEvent event)` This method is called when new `TVStream` is created
`void`	`streamDestroyed(TVConnectorEvent event)` This method is called when `TVStream` is destroyed

- **Method Detail**

- **connectDone**

```
void connectDone(TVConnectorEvent event)
```

This method is called when TVConnector connect attempt is done

Parameters:
`event` - the `TVConnectorEvent` indicating connect attemp is done

- **connectionDestroyed**

```
void connectionDestroyed(TVConnectorEvent event)
```

This method is called when TVConnector connection destroyed

Parameters:
`event` - the `TVConnectorEvent` indicating connection is destroyed

- **streamCreated**

```
void streamCreated(TVConnectorEvent event)
```

This method is called when new `TVStream` is created

Parameters:
`event` - the `TVConnectorEvent` indicating `TVStream` is created

- **streamDestroyed**

```
void streamDestroyed(TVConnectorEvent event)
```

This method is called when `TVStream` is destroyed

Parameters:
`event` - the `TVConnectorEvent` indicating `TVStream` is destroyed

6.3.6.5 Class TVStream

Video, audio TV stream.

Constructor and Description

`TVStream(java.lang.String description,org.kurento.client.RtpEndpoint endpoint)`
TV stream channel with media and control

- **Method Summary**

Modifier and Type	Method and Description
`void`	`addEventListener(TVStreamListener listener)` register event listener for stream events
`org.kurento.client.RtpEndpoint`	`getEndpoint()` get Kurento RTP endpoint
`int`	`getState()` get stream state
`void`	`requestHQ()` send request for high-quality media
`void`	`requestLive()` send **live** request for this stream, corresponding `TVStreamEvent` will be emitted
`void`	`requestPreview()` send **preview** request for this stream, corresponding `TVStreamEvent` will be emitted

- **Constructor Detail**

- **TVStream**
-
```
public TVStream(java.lang.String description,
                org.kurento.client.RtpEndpoint endpoint)
```

TV stream channel with media and control

Parameters:
`description` - description for stream
`endpoint` - kurento RTP endpoint

- ## Method Detail

- **addEventListener**

```
public void addEventListener(TVStreamListener listener)
```

register event listener for stream events

Parameters:
`listener` - `TVStreamListener` class implementation

- **getEndpoint**

```
public org.kurento.client.RtpEndpoint getEndpoint()
```

get Kurento RTP endpoint

Returns: `RtpEndpoint`

- **getState**

```
public int getState()
```

get stream state

Returns: current stream state

- **requestHQ**

```
public void requestHQ()
```

send request for high-quality media

- **requestLive**

```
public void requestLive()
```

send **live** request for this stream, corresponding `TVStreamEvent` will be emitted

- **requestPreview**

```
public void requestPreview()
```

send **preview** request for this stream, corresponding `TVStreamEvent` will be emitted

6.3.6.6 Class TVStreamEvent

This event is generated for a TV stream event. There are three main categories of TV stream events:

- Media stream events include media distribution (MMH)
- Stream control events include stream control via Central
- Management events include user or operator activity (moving to preview or Live)

Fields

Modifier and Type	Field and Description
`static int`	`LIVE_ENTERED` This event indicates that stream enter live mode
`static int`	`LIVE_LEAVED` This event indicates that stream leaved live mode
`static int`	`PREVIEW_ENTERED` This event indicates that stream enter preview mode
`static int`	`PREVIEW_LEAVED` This event indicates that stream leave preview mode
`static int`	`STREAM_CHANGED` This event indicates that stream quality changed
`static int`	`STREAM_CREATED` This event indicates that stream is created

Constructor and Description

`TVStreamEvent(int id, java.lang.String message)`
Initializes a new instance of `TVStreamEvent` with the specified information.

- **Field Detail**
  - **LIVE_ENTERED**

```
public static final int LIVE_ENTERED
```

This event indicates that stream enter live mode

  - **LIVE_LEAVED**

```
public static final int LIVE_LEAVED
```

This event indicates that stream leaved live mode

  - **PREVIEW_ENTERED**

```
public static final int PREVIEW_ENTERED
```

This event indicates that stream enter preview mode

  - **PREVIEW_LEAVED**

```
public static final int PREVIEW_LEAVED
```

This event indicates that stream leave preview mode

- **STREAM_CHANGED**

```
public static final int STREAM_CHANGED
```

This event indicates that stream quality changed

- **STREAM_CREATED**

```
public static final int STREAM_CREATED
```

This event indicates that stream is created

- **Constructor Detail**
- **TVStreamEvent**

```
public TVStreamEvent(int id, java.lang.String message)
```

Initializes a new instance of `TVStreamEvent` with the specified information. Invalid id leads to unspecified results.

Parameters:
`id` - the event id
`message` - event description

6.3.6.7 Interface TVStreamListener

This interface is for classes that wish to receive TV Stream events. This includes media,control, TV operator activity associated with TV Stream.

`void` `liveEntered(TVStreamEvent event)`
This method is called when TV Stream enter **live** mode

`void` `liveLeaved(TVStreamEvent event)`
This method is called when TV Stream leaves **live** mode

`void` `mediaStarted(TVStreamEvent event)`
This method is called when media (video and audio) streams is started

`void` `mediaStopped(TVStreamEvent event)`
This method is called when media (video and audio) streams is stopped

`void` `previewEntered(TVStreamEvent event)`
This method is called when TV Stream enter **preview** mode

`void` `previewLeaved(TVStreamEvent event)`
This method is called when TV Stream leaves **preview** mode

`void` `qualityChanged(TVStreamEvent event)`
This method is called when stream media quality is changed

o **Method Detail**

- **liveEntered**

```
void liveEntered(TVStreamEvent event)
```

This method is called when TV Stream enter **live** mode

Parameters:
`event` - the `TVStreamEvent` indicating entering in **live** mode

- **liveLeaved**

```
void liveLeaved(TVStreamEvent event)
```

This method is called when TV Stream leaves **live** mode

Parameters:
`event` - the `TVStreamEvent` indicating leaving from **live** mode

- **mediaStarted**

```
void mediaStarted(TVStreamEvent event)
```

This method is called when media (video and audio) streams is started

Parameters:
`event` - the `TVStreamEvent` indicating media stream is started

- **mediaStopped**

```
void mediaStopped(TVStreamEvent event)
```

This method is called when media (video and audio) streams is stopped

Parameters:
`event` - the `TVStreamEvent` indicating media stream is stopped

- **previewEntered**

```
void previewEntered(TVStreamEvent event)
```

This method is called when TV Stream enter **preview** mode

Parameters:
`event` - the `TVStreamEvent` indicating entering in **preview** mode

- **previewLeaved**

```
void previewLeaved(TVStreamEvent event)
```

This method is called when TV Stream leaves **preview** mode

Parameters:
`event` - the `TVStreamEvent` indicating leaving from **preview** mode

- **qualityChanged**

```
void qualityChanged(TVStreamEvent event)
```

This method is called when stream media quality is changed

Parameters:
`event` - the `TVStreamEvent` indicating media stream quality changed

7 References

[ALVESTRAND] https://tools.ietf.org/html/draft-ietf-rtcweb-overview-14
[BOOST] http://www.boost.org
[FMC] http://www.f-m-c.org
[GObject] https://en.wikipedia.org/wiki/GObject
[GSTREAMER] http://gstreamer.freedesktop.org/
[HARRIS1998] A combined corner and edge detector, Chris Harris & Mike Stephens. http://courses.daiict.ac.in/pluginfile.php/13002/mod_resource/content/0/References/harris1988.pdf
[JSONCPP] https://github.com/open-source-parsers/jsoncpp
[KURENTO] http://www.kurento.org
[LGPL] http://en.wikipedia.org/wiki/GNU_Lesser_General_Public_License
[GSTDOT] https://developer.ridgerun.com/wiki/index.php/How_to_generate_a_Gstreamer_pipeline_diagram_%28graph%29
[OPENCV] OpenCv library, http://opencv.org/.
[RFC0768] https://www.rfc-editor.org/rfc/rfc768.txt
[RFC3550] https://www.ietf.org/rfc/rfc3550.txt
[RFC3551] https://www.ietf.org/rfc/rfc3551.txt
[RFC4566] https://tools.ietf.org/html/rfc4566
[RFC4585] https://tools.ietf.org/html/rfc4585
[TRICKLE ICE] https://tools.ietf.org/html/draft-ietf-mmusic-trickle-ice-02
[UML] http://www.uml.org/
[VIOLACVPR] Rapid Object Detection using a Boosted Cascade of Simple Features. Paul Viola and Michael Jones. https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-cvpr-01.pdf
[WEBTCOV] https://tools.ietf.org/html/draft-ietf-rtcweb-overview-14