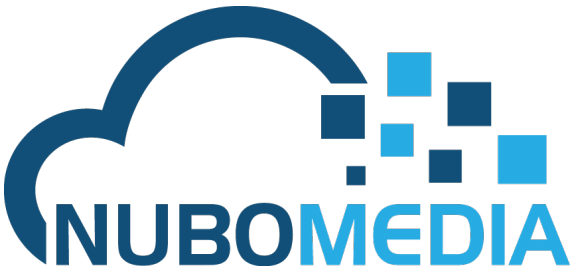
D5.2	
Version	1.3
Author	URJC
Dissemination	PU
Date	31/01/2016
Status	Final



D5.2: NUBOMEDIA framework APIs and tools v1

Project acronym:	NUBOMEDIA
Project title:	NUBOMEDIA: an elastic Platform as a Service (PaaS) cloud for interactive social multimedia
Project duration:	2014-02-01 to 2017-01-31
Project type:	STREP
Project reference:	610576
Project web page:	http://www.nubomedia.eu
Work package	WP5
WP leader	Luis López
Deliverable nature:	Report
Lead editor:	Luis López
Planned delivery date	01/2015
Actual delivery date	31/01/2016
Keywords	NUBOMEDIA, Kurento, Application Programming Interface

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:

Luis Lopez (URJC)
Boni García (URJC)
Mondin Fabio Luciano (TI)
Mäkelä Satu-Marja (VTT)
Jukka Ahola (VTT)
Samuli Heinonen (VTT)
Teofilo Redondo (ZED)

Internal Reviewer(s):

Ivan Gracia (NAEVATEC)

Version History

Version	Date	Authors	Comments
1.0	03-01-2015	Luis Lopez	Initial version
1.1	13-01-2016	Boni García	Reviewed version
1.2	21-01-2016	Boni García	Integrated contributions

Table of contents

List of Figures:

Acronyms and abbreviations:

API	Application Programming Interface
AR	Augmented Reality
CDN	Contend Distribution Network
FOSS	Free Open Source Software
IMS	IP Multimedia Subsystem
IoT	Internet of Things
KMS	Kurento Media Server
MCU	Multipoint Control Unit
NFV	Network Function Virtualization
RTC	Real-Time Communications
RTP	Real-time Transport Protocol
SCTP	Stream Control Transmission Protocol
SFU	Selective Forwarding Unit
UE	User Equipment
VCA	Video Content Analysis
VoD	Video on Demand
WebRTC	Web Real Time Communications

1 Executive summary

This document presents a detailed description of the different media APIs (Application Programming Interfaces) developed in the context of the project NUBOMEDIA, namely:

- The **NUBOMEDIA Media API**. It consists on a server-side API that exposes media capabilities through pipelining mechanisms. This API is built on the top of two main concepts: Media Element (holder for a specific media capability) and Media Pipeline (graph of connected Media Elements). Media Elements are like Lego pieces: developers need to take the elements needed for an application and connect them following the desired topology. Hence, when creating a pipeline, developers need to determine the capabilities they want to use (the media elements) and the topology determining which media elements provide media to which other media elements (the connectivity).
- The **NUBOMEDIA Repository API**. This API has the objective of exposing the capability of storing and recovering multimedia data (and metadata) in a cloud environment in a scalable, reliable and secure way. In addition, it provides mechanisms to enable media interoperability with the NUBOMEDIA Media API, so that specific Media Elements are able to record/play media in/from the repository.
- The **NUBOMEDIA WebRtcPeer API**. This API is focused on WebRTC, wrapping the W3C's Media Capture and the PeerConnection API. In short, it proposes a seamless easy API to work with WebRTC streams in the client-side, hiding the complexity of handling users' audio and video and media negotiation.
- The **NUBOMEDIA Signaling API**. This API implements a full duplex channel between clients and servers following a request/response mechanism. To that aim, it uses JSON-RPC v2 over WebSockets.
- The **NUBOMEDIA Room API**. This API has the objective of enabling application developers to create group communication applications adapted to real social interactions. This API has several layers, involving a room SDK, a server-side and two client-side APIs. It handles two main entities: rooms (groups of peers that are able to communicate each other by means of WebRTC) and participants (users within the room).
- The **NUBOMEDIA Tree API**. This API allows developers to build WebRTC broadcasting. It has been implemented as a server-side service, and therefore there are clients that consume this service. In other words, there are clients feeding the broadcasting with WebRTC media, while a big number of different clients consume that media.

2 Introduction

The main objective of the NUBOMEDIA project is to simplify the task of creating applications involving real-time media capabilities. For this, the project research and development efforts are split into two main types of activities: one side we have the tasks devoted to creating technological enablers such as cloud infrastructures, PaaS mechanisms and media capabilities; on the other, we have the efforts devoted at exposing such enablers to developers through a set of APIs. This document accounts for this latter type.

Creating APIs is a complex task for which no well established methodologies exist. It is typically based on designers' intuition and on iterative cycles validation cycles in which

developers use the APIs for creating applications and services and, during the process, detect their limitations, drawbacks and problems. These latter are fixed and the iterative process continues.

This document presents the first iteration of this process for the NUBOMEDIA APIs. These APIs have been designed following a top-down approach in which the specific requirements generated by the NUBOMEDIA partners have been taken into consideration at the time of determining which APIs are created and what are their features.

This document presents these APIs basing on the following structure. Fist, the objective of the document is presented to show that it is fully aligned with the project objectives. Second, we introduce our API architecture, which is based on a layered approach where APIs are stacked among each other following increasing levels of abstraction. Each of these API layers is presented through a number of guidelines showing each API objectives, scope and capabilities. To conclude, some implementation details are introduced.

3 Objectives

This document accounts for the activities carried out in the context of Work Package 5 of the NUBOMEDIA project. Due to this its main objective is to describe how the different NUBOMEDIA APIs have been created and how they can be leveraged in the context of the NUBOMEDIA project. This objective can be split into a number of sub-objectives, which are the following:

- To describe the NUBOMEDIA APIs as a stack of APIs adapted to developers with different profiles and degrees of expertise so that higher level APIs are more abstract and simple to use than low level APIs.
- To describe the NUBOMEDIA Media API as a modular API basing on the concept of media pipelines: chain of media elements which interconnect different capabilities through dynamic topologies under the control of developers.
- To describe how high-level APIs have been ported in all common popular platforms including WWW browsers and smartphone native clients.
- To specify how the API stack has been structured as Free Open Source Software so that it can be leveraged by the NUBOMEDIA community and ecosystem.

4 NUBOMEDIA APIs architecture an overview

4.1 NUBOMEDIA framework API architecture

NUBOMEDIA is a Platform as a Service cloud. It has been conceived for making simple and efficient the development and exploitation of applications involving Real-Time media Communications (RTC). To this aim, NUBOMEDIA capabilities are exposed through simple APIs which try to abstract all the low level details of service deployment, management, and exploitation allowing applications to transparently scale and adapt to the required load while preserving QoS guarantees.

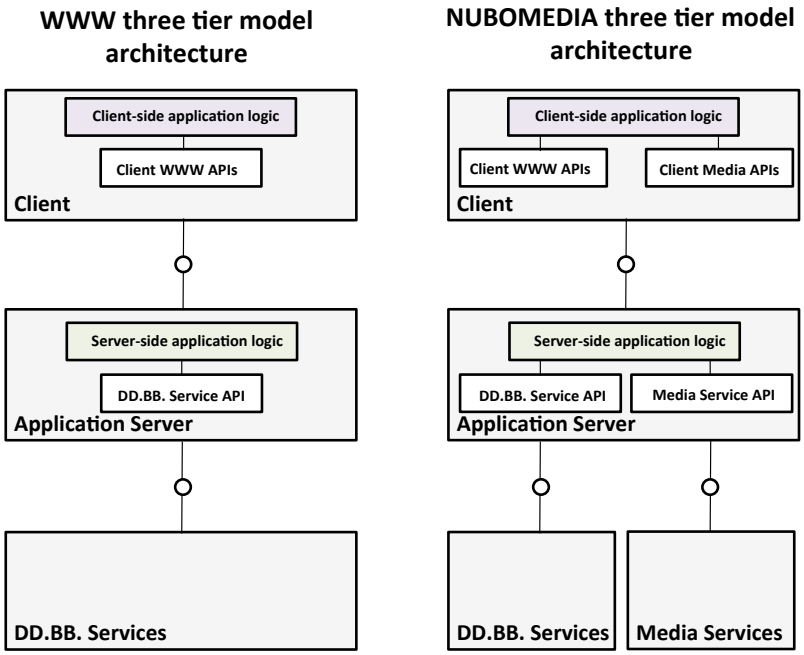


Figure 1. On the left, we depict the typical architecture of a WWW application basing on the three tier model. On the right, we depict the conceptual representation of a NUBOMEDIA application. As it can be observed, the only difference is that a NUBOMEDIA application has access to additional media-oriented APIs, which does not restrict developer's freedom for using the additional technologies or services she wishes.

From the developer's perspective, NUBOMEDIA capabilities are accessed through a set of APIs inspired by the popular WWW three tier model (see Figure 1). In this model, WWW developers distinguish three layers:

- The Client, which consists on a WWW browser executing the client-side application logic. As WWW browsers are typically used as thin clients, this just contains the logic for controlling the presentation and end-user interaction mechanisms. This logic is typically developed using programming languages such HTML and JavaScript; and with the help of specialized third party APIs (e.g. Angular, Bootstrap, etc.)
- The Application Server (AS), which hosts the server-side application logic. This layer typically contains the business logic of the application. Hence, aspects such as how end-user information is managed, what are the allowed application workflows or which users have access to which capabilities are controlled by this layer. Very typically, for developing this logic developers consume third party APIs enabling access to the service layer capabilities.
- The Service Layer comprise a number of services that are required for the application to work. These very often consist on DD.BB. and communication services.

NUBOMEDIA has been designed basing on this model: in the same way that WWW developers program their business logic consuming Data Base (DD.BB.) APIs, NUBOMEDIA makes possible to create rich media services through media-oriented APIs. Hence, NUBOMEDIA embraces the WWW development model and the WWW development methodology, which is quite convenient for the millions of WWW developers out there.

As a result, and as it can be observed in Figure 1, when creating NUBOMEDIA applications, the only difference that a WWW developer finds is the availability of additional media-oriented APIs. These enable accessing client-side RTC and server-side

RTC media capabilities. Clearly, this does not restrict developers' freedom for architecting their applications basing on the technologies and additional APIs of their preference.

Hence, for leveraging NUBOMEDIA capabilities, developers just need to understand the NUBOMEDIA APIs so that they may use them for creating their RTC media applications. The NUBOMEDIA API stack is architected following the scheme depicted in Figure 2. As it can be observed, this stack offers a complete set of APIs that can be classified in three groups that we call: Media Capabilities APIs, Signaling APIs and Abstract Communication APIs.

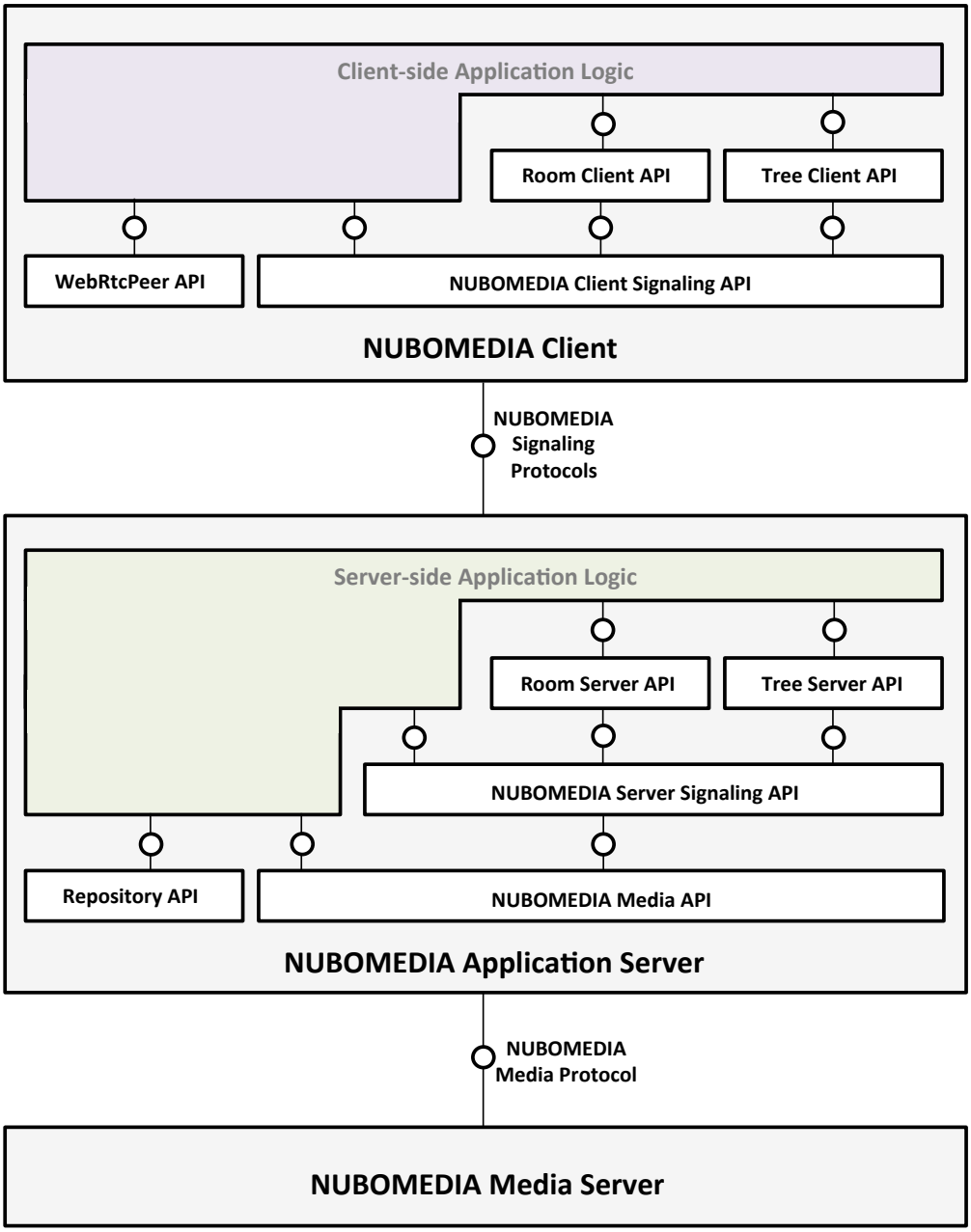


Figure 2. Architectural diagram showing the NUBOMEDIA API stack which comprises three types of APIs: Media Capability APIs (NUBOMEDIA Media and Repository APIs), Signaling APIs (JSON-RPC over WebSocket client and server APIs), and Abstract Communication APIs (Room and Tree client and server APIs)

4.1.1 Media Capabilities APIs

These APIs expose to the application logic the low-level media capabilities of NUBOMEDIA. These capabilities are basically the ones offered by the NUBOMEDIA Media Server (as described in NUBOMEDIA Project Deliverable 4.2) and the NUBOMEDIA Media Repository (as described in NUBOMEDIA Project Deliverable 3.2). As a result, this group includes two APIs:

- The NUBOMEDIA Media API, which enables developers consuming the NUBOMEDIA Media Server capabilities among which we can find media transport, media archiving, media processing, media transcoding, etc. This API is based on two main concepts: Media Elements and Media Pipelines.
- The NUBOMEDIA Repository API, which makes possible to access an elastic scalable media repository for archiving media information and meta-information. It is based on the notion of Repository Item: an object with a unique identity which may contain media data and meta-data.

In addition, in this group we also include the NUBOMEDIA API abstracting the client side media capabilities for developers. In current state-of-the-art, these capabilities basically correspond to WebRTC and comprise both the ability of capturing and rendering media plus the ability of communicating RTC media. In NUBOMEDIA, we have abstracted all these capabilities behind the WebRtcPeer API.

4.1.2 Signaling APIs

The Media Capabilities APIs introduced above are signaling agnostic, meaning that they do neither require nor assume any kind of specific characteristic for signaling. Hence, NUBOMEDIA capabilities can be accessed through any kind of signaling protocol including SIP, XMPP or REST. However, for the sake of simplicity, we have created a very simple signaling protocol based on JSON-RPCs over WebSockets that is suitable for most applications not requiring specific interoperability features. This protocol has been wrapped through an abstract API which enables full-duplex signaling exchanges between Application Server and Clients. To this aim, the API is split into two complementary parts:

- The JSON-RPC over WebSocket client API. This is a client-side API enabling the creation of JSON-RPC sessions and the sending and receiving of messages through them.
- The JSON-RPC over WebSocket server API. This is a server-side API enabling the creation of JSON-RPC sessions and the implementation of server-side business logic for messages following a request/response communication scheme.

4.1.3 Abstract Communication APIs

Developers typically use RTC media capabilities for creating applications devoted to person-to-person communications. In this case, the application business logic needs to manage the communication topologies among participants in an RTC multimedia session. In the general case, this logic needs to be programmed by the application developers. However, there are a number of communication topologies that are quite popular and appear systematically in applications. Due to this, we have created specific communication API abstracting the low level details of the media logic on these topologies so that developers may use them in an agile and efficient manner. In particular, we have identified two of these common topologies: the Room Topology and the Tree Topology. For them two specific APIs have been implemented:

- The Room API. This API has two complementary sides, the Room Server and the Room Client APIs. Through them, this API makes possible for developers to

create applications basing on Room Topologies. Room topologies are based on the intuitive idea of a physical room where people may communicate. If you enter into the room you may talk to others and others may talk to you. Hence, the main characteristic of a Room Topology is that the multimedia session enables a full-duplex communication exchange of media among all participants. Hence, the session is called informally room and all room members may publish their media streams to the room or subscribe to the streams of the rest of room members.

- The Tree API. Like the Room API, it also has Server and Client sides. However, this API is based on a Tree Topology. In our context, a tree can be sees as a one-way (i.e. simplex) mechanism providing one-to-many RTC media. The interesting aspect of this topology is that the number of leafs of the tree may be large, which makes convenient for providing real-time media broadcasting capabilities to large audiences.

4.1.4 The NUBOMEDIA API stack rationale

The NUBOMEDIA API stack architecture depicted in Figure 2 is not random but it is the result of a complex design process where we have prosecuted to enable simplicity without scarifying flexibility and expressiveness. This stack has been created for abstraction, understood as the ability of making simple things simple and complex things possible for developers. For this, the rationale behind its design is based on a number of principles application developers need to understand in order to choose the appropriate APIs for each implementation objective.

In NUBOMEDIA, the most abstract APIs are the Room and Tree APIs. These are abstract because they hide most of the low level details of the media complexities and expose to developers high-level notions such as “Rooms” and “Trees”, that are logical objects suitable for managing media transport through specific communication topologies. The NUBOMEDIA internal use-cases analysis (see NUBOMEDIA Project Deliverable D2.1.2), as well as the accumulated experience of developers worldwide out of NUBOMEDIA evidence that Room and Tree topologies are at the base of a significant fraction of RTC media services. Hence, developers wishing to create applications just providing room group communications or tree one-to-many media broadcasting services can use these APIs directly without requiring any further understanding on the rest of the API stack.

However, there is still a fraction of developers for which the Room and Tree APIs do not fit. These may include the ones with specific interoperability requirements (e.g. SIP or legacy RTP) or needing special features (e.g. custom media processing, non-common communication topologies, non-linear media logic, etc.). In that case, lower-level NUBOMEDIA APIs might be needed.

The NUBOMEDIA Signaling APIs makes possible for developers to create custom signaling protocols in a seamless and efficient way. Hence, this API might be useful whenever specific signaling mechanisms beyond rooms and trees are required. Just for illustration, let’s imagine a video surveillance application with the ability of detecting intruders in a specific area of the camera viewport. Whenever an intruder is detected an alarm needs to be fired to all the connected clients, so that they may rewind the streams and visualize it again for assessing the severity of the incident.

Clearly, this type of logic cannot be provided through the Room or Tree APIs, which do not have alarm-sending capabilities or the ability of seeking the media for a playback.

Hence, developers creating this application need custom signaling protocols. These developers may use the signaling protocol they wish given the PaaS nature of NUBOMEDIA. However, among the available options, a natural choice is the NUBOMEDIA Signaling API. This API makes possible to create a custom signaling protocol just be defining some simple JSON-RPC messages, which is quite convenient given the familiarity of developers with this format. Once this is done, the API makes straightforward to send such messages and to implement the appropriate business logic upon their reception. The only limitation of this scheme is that the API mandates the protocol to be based upon JSON-RPC over WebSocket transport. Hence, whenever this restriction is not an impediment, the NUBOMEDIA Signaling API shall be useful for crating specific and customized signaling mechanisms.

The NUBOMEDIA Media API, in turn, exposes the low level media capabilities. Through this API developers can create arbitrary and dynamic topologies combining them with media processing, transcoding or archiving. The Room and Tree topologies are just particular cases of what this API can provide. Hence, mastering this API is a must for all developers wishing to take advantage of all NUBOMEDIA features. This API is complemented with the Repository API which enables media data and metadata persistence onto elastic scalable repositories. Hence, the Repository API may be of help whenever large amounts of media information need to be recorded and recovered.

All in all, and as Figure 2 shows, developers are free to combine the NUBOMEDIA APIs without restrictions so that for example, an application may consume at the same time the Room API, the Signaling API and the Media API if it is needed.

4.2 The NUBOMEDIA Media API

4.2.1 Objectives

The NUBOMEDIA Media API has the objective of exposing NUBOMEDIA Media Capabilities through a pipelining mechanism. This objective can be split into a number of sub-objectives, which are the following:

- To design and implement a Media Element abstraction suitable for exposing to application developers different types of media capabilities including media transport, media encoding/decoding, media processing and media archiving.
- To design and implement a Media Pipeline abstractions suitable for creating graphs of Media Elements providing custom media processing logic.
- To enable the API to extend the notion of multimedia so that it becomes more than audio-visual information. This means that the API may enable developers to manage arbitrary multi-sensory information as part of the RTC media stream.
- To enable developers to monitor QoS and QoE metrics so that applications can be improved and optimized accordingly to developer's needs.
- To make possible to comply with all the above specified objectives in a PaaS cloud environment.

4.2.2 Scope

The NUBOMEDIA Media API is a server-side API. This means that it is only of relevance for developers wishing to create custom media control logic to execute into application servers. Hence, this API is not of relevance for client-side developers.

The NUBOMEDIA Media API has been designed for the sole purpose of providing media control capabilities to application developers basing on the notion of NUBOMEDIA Media Element and Media Pipeline. This API does not assume any

specific requirements on the underlying implementation of the south bound media driver (i.e. the media server control protocol and the media capabilities themselves.)

This API is fully agnostic to signaling. This means that it does neither provide nor assume any kind of characteristic or feature from the signaling protocol. In particular, this API does not provide any kind of call control mechanism or logic. This API does not provide either any kind of AAA (Authentication, Authorization and Accounting) facility. Hence, security requirements must be fully implemented at the application logic.

Media Server capabilities are exposed by the NUBOMEDIA Media API to application developers. Different types of clients can consume this API. Figure 3 shows an example of different clients consuming the NUBOMEDIA Media API:

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

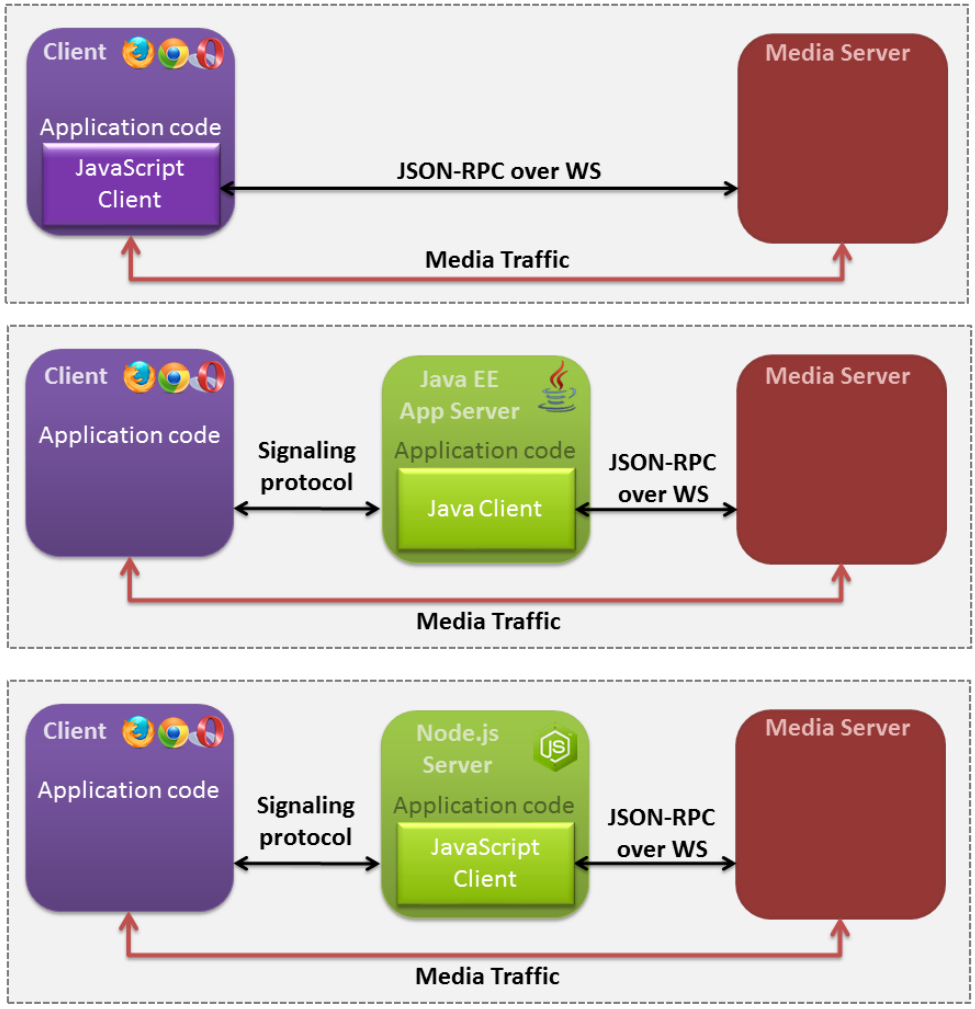


Figure 3. Schema of different kind of applications using the NUBOMEDIA Media API

4.2.3 API Overview

The NUBOMEDIA Media API is built on top of an object oriented model where the root of the inheritance hierarchy is the `MediaObject`. The `MediaObject` is only a holder providing utility members (it is abstract and cannot be instantiated). The two

main types inheriting from `MediaObject` are `MediaElement` and `MediaPipeline`.

The `MediaElement` is the main abstraction of the NUBOMEDIA Media API. Intuitively, a `MediaElement` can be seen as a black box implementing a specific media capability. In general, `MediaElements` receive media streams through sinks, send media streams through sources and, in the middle, do "something" with the media. There are two main subclasses of `MediaElements`: Endpoints and Filters. An Endpoint is always a `MediaElement` with the ability of communicating media with the external world. All media streams coming into an Endpoint sink are send out of the `MediaElement` through some kind of external interface (e.g. network interface, file system interface, etc.) In the same way, all media streams received from the external interface are published and made available to other `MediaElements` through the Endpoint source. Filters, on the other hand, do not communicate media streams with the external world. Their only function is to implement some kind of media processing. This can be simple transport (e.g. a pass-through filter) or may involve complex processing algorithms including computer vision or augmented reality.

`MediaElements` can be connected among each other through a `connect` primitive. When a `MediaElement` (let's call it A) is connected to other `MediaElement` (say B), the media streams available at A's source are feed to B's sink. The connectivity of `MediaElements` works following quite intuitive and natural rules. First, a `MediaElement` source can be connected to as many `MediaElement` sinks as you want (i.e. a `MediaElement` can provide media to many `MediaElements`). Second, a `MediaElement` sink can only receive media from a connected source. Hence, connecting a source to a sink that is previously connected makes that sink to first disconnect from its previous source before being connected to the new one. Hence, application developers create their media processing logic just by connecting media elements following the desired topology.

`MediaPipelines`, in turn, are just containers of `MediaElement` graphs. A `MediaPipeline` holds `MediaElements` that can connect among each other following an arbitrary and dynamic topology. `MediaElements` owned by one `MediaPipeline` cannot connect to `MediaElements` owned by another `MediaPipeline`. Hence, the `MediaPipeline` represents and isolated multimedia session from the perspective of the application.

4.2.4 Features

The features provided by the NUBOMEDIA Media API comprise specific media capabilities that are made available to application developers to create their media enabled applications. These capabilities can be grouped into two main categories: media elements, which inherit from the `MediaElement` class and manage a single media stream, and hubs, which inherit from the `Hub` class and have been specifically designed for the management of groups of streams.

Media elements have two flavors: Endpoints and Filters. Endpoints are in charge of the I/O media operations in the media pipeline. Figure 4 shows the NUBOMEDIA Media API endpoint inheritance hierarchy, which comprises the following capabilities:

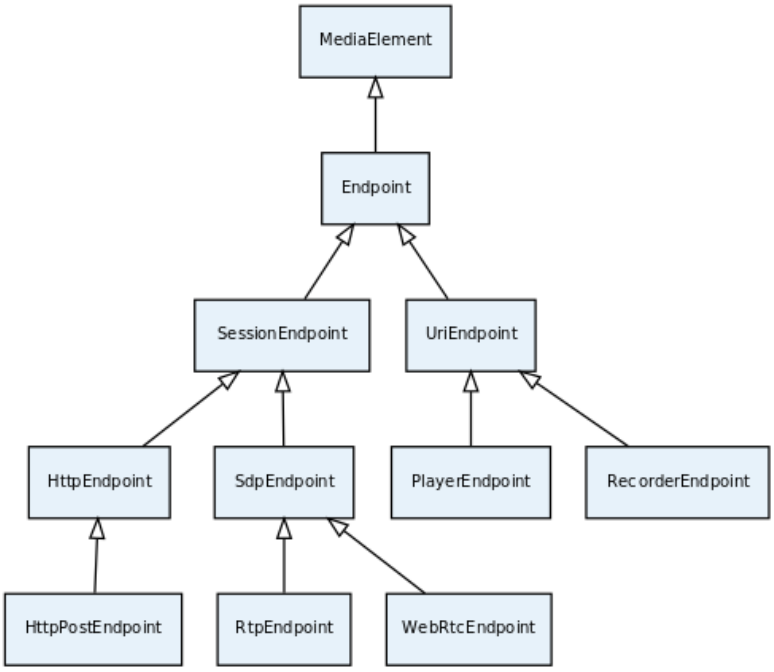


Figure 4. UML class diagram of the Endpoints specified by the NUBDOMEDIA Media API

- The `WebRtcEndpoint` is an I/O endpoint that provides full-duplex WebRTC media communications compatible with the corresponding protocol standard. It is important to remark, that among `WebRtcEndpoint` capabilities the NUBOMEDIA Media API defines as mandatory the DataChannel support. DataChannels are a mechanism for receiving media information beyond audio and video given their ability to accommodate arbitrary sensor data that is transported in the same ICE (Interactive Connectivity Establishment) connection than the audio and the video and, hence, may maintain synchronization with them.
- The `RtpEndpoint` is equivalent but with the plain RTP protocol.
- The `HttpPostEndpoint` is an input-only endpoint that accepts media using HTTP POST requests. This capability needs to support HTTP multipart and chunked encodings, so that it is compatible with the HTTP file upload function exposed by WWW browsers. This endpoint must support the MP4 and WebM media formats.
- The `PlayerEndpoint` is an input-only endpoint that retrieves content from the local file system, HTTP URLs or RTSP URLs and injects it into the media pipeline. This endpoint must support the MP4 and WebM media formats for all input mechanisms as well as RTP/AVP/H.264 for RTSP streams.
- The RecorderEndpoint is an output-only endpoint that provides function to store contents in reliable mode (doesn't discard data). This endpoint may write media streams to the local file system, or to HTTP URLs using POST messages. This endpoint must support MP4 and WebM media formats.

Filters, in turn, are used for processing media streams. Filters are useful for integrating different types of capabilities such as Video Content Analysis (VCA), Augmented Reality (AR) or custom media adaptation mechanisms. The NUBOMEDIA Media API does not specify any kind of mandatory filter and it is let to API implementers to define their filters following the NUBOMEDIA Media API extensibility mechanisms.

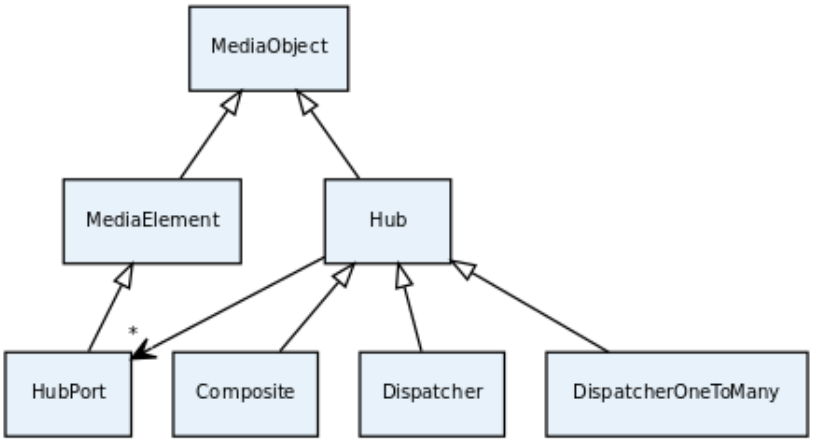


Figure 5. UML class diagram of main Hub types in the NUBOMEDIA Media API

Hubs follow the inheritance scheme depicted in Figure 5. `Hubs` work in coordination with `HubPorts`: a special type of media element, which provides sinks and sources to hubs. The NUBOMEDIA Media API defines the following types of hubs:

- `Composite` is a hub that mixes the audio stream of its connected inputs and constructs a grid with the video streams of them.
- `Dispatcher` is a hub that allows routing between arbitrary input-output HubPort pairs.
- `DispatcherOneToMany` is a Hub that sends a given input to all the connected output HubPorts.

4.2.5 Example use cases

The typical use cases that can be implemented using the NUBOMEDIA Media API involve the following aspects:

1. Real time communications in the Web and smartphones platforms. As described before, the NUBOMEDIA Media API provides the capability to work with WebRTC streams.
2. Recording capabilities. Another important feature of the NUBDOMEDIA Media API is the ability to record media.
3. Interoperating with IP cameras (e.g. video surveillance). Thanks to the capability to handle RTP streams within the media server, the stream from IP cameras systems can be easily handled with the NUBOMEDIA Media API.
4. Computer vision or augmented reality capabilities. As introduced in section before, filters are the elements of the NUBOMEDIA Media API than can be used to process media streams. Some examples of these capabilities are: face recognition, crowd detection, and QR and bar code detection within a media stream.

To illustrate these concepts, let's see a simple application. This application performs a full-duplex back-to-back call between two users and records their streams into a repository. Figure 6 shows the corresponding pipeline.

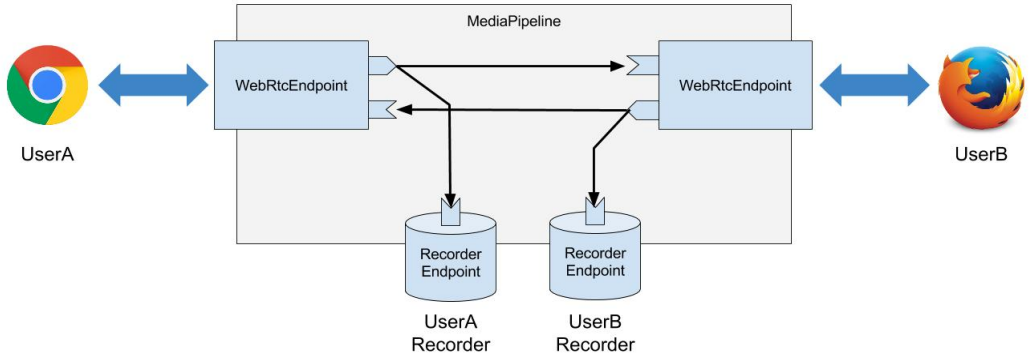


Figure 6. Architectural diagram of an example application performing a back-to-back call between to users where their corresponding streams are recorded

This pipeline can be implemented in Java with the code shown in the following snippet:

```
KurentoClient rtcMediaAPI = KurentoClient.create();

MediaPipeline pipeline = rtcMediaAPI.createMediaPipeline();

WebRtcEndpoint userA = new WebRtcEndpoint.Builder(pipeline).build();
WebRtcEndpoint userB = new WebRtcEndpoint.Builder(pipeline).build();

userA.connect(userB);
userB.connect(userA);

RecorderEndpoint userARecorder =
    new RecorderEndpoint.Builder(pipeline,"videoUserA.webm").build();

RecorderEndpoint userBRecorder =
    new RecorderEndpoint.Builder(pipeline,"videoUserB.webm").build();

userA.connect(userARecorder);
userB.connect(userBRecorder);
```

4.2.6 API availability

The NUBOMEDIA Media API is available in the following programming languages

- Java v6.0 or later.
- JavaScript for Node.js
- JavaScript for browser

4.2.7 Information for application developers

Once the main features of the API have been introduced, this section is devoted to providing to application developers all the information enabling to create applications using it. This section is designed as a collection high level explanations, pointers and links towards the appropriate developer guidelines and reference information.

Important notice

NUBOMEDIA media capabilities are provided by Kurento Media Server. As a result, the Kurento media architecture is inherited by NUBOMEDIA. This has a major consequence for developers: the NUBOMEDIA Media API has been implemented as an extension of the Kurento Client API. This means that all documentation, tutorials and knowledge involving Kurento Client API can be directly applied to NUBOMEDIA. The following aspects need to be taken into consideration:

- NUBOMEDIA Media API developers MUST use the kurento-client-extended, as described in NUBOMEDIA Project Deliverable D3.2 Java SDK, which makes agnostic media server autoscaling and media pipeline scheduling mechanisms thanks to NUBOMEDIA NFV services. Hence, the kurento-client-extended SDK is the software artifact exposing the NUBOMEDIA Media API.
- Plain Kurento developers, which cannot enjoy autoscaling, MAY use the kurento-client Java SDK or the kurento-client JavaScript SDK.

Tutorials and developer guides
Tutorials showing how to obtain and use the kurento-client-extended SDK can be found in the following links

- https://github.com/fhg-fokus-nubomedia/kurento-client-extended

Tutorials showing how to use the media capabilities exposed through the kurento-client SDK are available at the official Kurento Client API documentation, which can be found here

- Kurento official documentation
  - http://doc-kurento.readthedocs.org/en/stable/

In particular, Java tutorials from 1 to 5 are relevant for understanding how to create applications basing on the NUBOMEDIA Media API, these can be obtained on the following URLs:

- Tutorial 1. Hello World. This web application has been designed to introduce the principles of programming with Kurento for Java developers. It consists on a WebRTC video communication in mirror (loopback).
  - http://doc-kurento.readthedocs.org/en/stable/tutorials/java/tutorial-1-helloworld.html
- Tutorial 2. WebRTC magic mirror. This web application extends tutorial 1 adding media processing (concretely face recognition) to the basic WebRTC loopback.
  - http://doc-kurento.readthedocs.org/en/stable/tutorials/java/tutorial-2-magicmirror.html
- Tutorial 3. One to many video call. This web application consists on a one-to-many video call using WebRTC technology. In other words, it is an implementation of a video broadcasting web application.
  - http://doc-kurento.readthedocs.org/en/stable/tutorials/java/tutorial-3-one2many.html
- Tutorial 4. One to one video call. This web application consists on a one-to-one video call using WebRTC technology. In other words, this application provides a simple video softphone.
  - http://doc-kurento.readthedocs.org/en/stable/tutorials/java/tutorial-4-one2one.html
- Tutorial 5. Advanced one to one video call. This web application consists on an advanced one to one video call using WebRTC technology. It is an improved version of the tutorial 4).
  - http://doc-kurento.readthedocs.org/en/stable/tutorials/java/tutorial-5-one2one-adv.html

Reference information
Reference information providing the Javadoc of the Kurento Client API can be found here:

- Kurento Client official Javadoc reference.
  - http://doc-kurento.readthedocs.org/en/stable/_static/langdoc/javadoc/index.html

Source code and licensing
The repositories containing the relevant artifacts involved in this API are the following:

- kurento-client-extended SDK
  - Source code
    - https://github.com/fhg-fokus-nubomedia/kurento-client-extended
  - License
    - LGPL v2.1
- kurento-client SDK
  - Source code
    - https://github.com/Kurento/kurento-java/tree/master/kurento-client
  - License
    - LGPL v2.1

4.3 The NUBOMEDIA Repository API

4.3.1 Objectives

This API has the objective of exposing the NUBOMEDIA Repository capabilities to application developers. This can be split into the following sub-objectives:

- To expose the capability of storing and recovering multimedia data in a cloud environment in a scalable, reliable and secure way.
- To expose the capability of storing and recovering multimedia metadata associated to the above mentioned multimedia data.
- To provide the appropriate mechanism enabling media interoperability with the NUBOMEDIA Media API, so that specific Media Elements are able to record/play media in/from the repository
- To make possible to comply with the above specified objectives in a PaaS cloud environment.

4.3.2 Scope

The NUBOMEDIA Repository API has been designed for interoperating with the NUBOMEDIA Media Capabilities. In particular, it has been designed for enabling Media Pipelines to store and recover media from the NUBOMEDIA Repository. To this aim, this API complements the capabilities of two specific types of Media Elements:

- The `RecorderEndpoint`, so that this API makes possible a `RecorderEndpoint` to store multimedia data into the NUBOMEDIA Repository.
- The `PlayerEndpoint`, so that this API makes possible for a `PlayerEndpoint` to recover multimedia data from the NUBOMEDIA Repository.

4.3.3 API Overview and features

The Repository API is designed around the concept of Repository Items. A Repository Item is a logical object in the repository which has three distinguishing characteristics:

- It has a unique ID, called `itemId` or just `Id`.

- It has associated media data.
- It has associated media metadata.

The Repository API makes possible to manage Repository Items through CRUD (Create, Read, Update and Delete) operations with one exception: media data cannot be updated. This means that, once the Repository Item media data has been stored it becomes read only and any update on it requires the creation of a whole new Repository Item.

Repository Items are managed through thee `RepositoryClient` interface, which exposes a number of utility methods to that aim. Without loss of generality, in the table below we specify this interface in the Java Programming language. Other language bindings could be also created:

```
public interface RepositoryClient {
  RepositoryItemRecorder createRepositoryItem(Map<String, String> metadata);
  Response removeRepositoryItem(String itemId);
  RepositoryItemPlayer getReadEndpoint(String itemId);
  Set<String> simpleFindItems(Map<String, String> searchValues);
  Set<String> regexFindItems(Map<String, String> searchValues);
  Map<String, String> getRepositoryItemMetadata(String itemId);
  Response setRepositoryItemMetadata(String itemId,
                                     Map<String, String> metadata);
}
```

For completeness, let's explain the semantics of each the most important of those methods:

`RepositoryItemRecorder createRepositoryItem(Map<String, String> metadata);`
This method is used for creating a new repository item. As it can be observed, at creation time, we can specify custom metadata as key/value pairs. As a result of invoking this method a `RepositoryItemRecorder` is returned. This is a specific type representing a Repository Item where media data can be recorded. This type has only two attributes: `id`, which provides the newly created item ID, and `url`, which provides the HTTP URL where the media data to be recorded should be sent (i.e. the URL to be provided to the `RecorderEndpoing` issuing the media to be recorded).

`RepositoryItemPlayer getReadEndpoint(String itemId);`
This method is used for recovering a Repository Item through ID. The return value is a type representing a Repository Item ready for reading. As such, it holds an `url` attribute where the media data associated with the Item can be read (i.e. the URL to be provided to the corresponding `PlayerEndpoing` getting media into a Media Pipeline).

`Set<String> simpleFindItems(Map<String, String> searchValues);`
`Set<String> regexFindItems(Map<String, String> searchValues);`
These two methods make possible to recover the IDs of repository items containing a specific metadata key/value pair (the former) or having metadata values associated to specific keys to match with a given regular expression (the latter).

`Map<String, String> getRepositoryItemMetadata(String itemId);`
`Response setRepositoryItemMetadata(String itemId,`
`Map<String, String> metadata);`
These two methods make possible to read or update the metadata of a given repository item, specified by ID.

4.3.4 Example use cases

The typical use case involving the NUBOMEDIA Repository API comprises two steps:

1. Creation of repository item and upload of media
2. Query repository through id and consumption of the media

The following picture illustrated these two steps in a flow diagram:

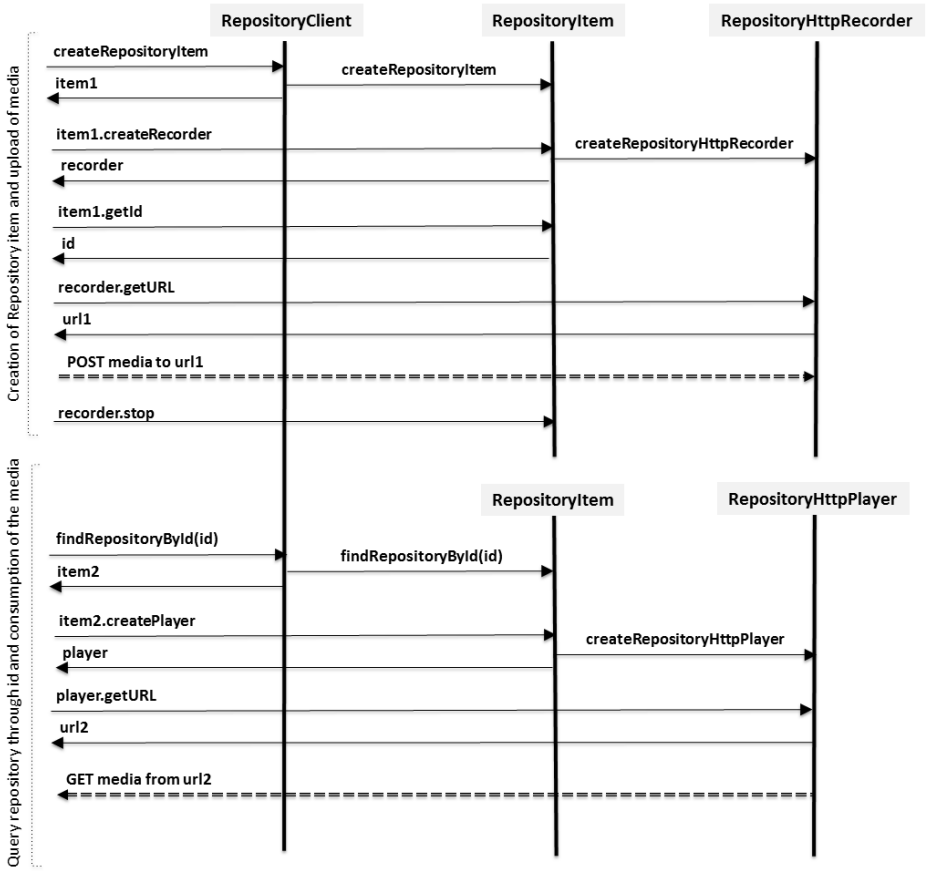


Figure 7. Flow diagram of the main use cases of the NUBOMEDIA Repository API

As can be seen in this flow diagram, to create a repository item the first step is requesting the creation of a `RepositoryItem` to a `RepositoryClient`. In order to upload media to the repository, a `RepositoryHttpRecorder` should be created using the `RepositoryItem`. Then, media can be uploaded using the method POST of the HTTP protocol. This mechanism can be used to interact with the NUBOMEDIA Media API. The Media Element `RecorderEndpoint` can be feed by means of an URL in which media is uploaded by POST.

Once the media upload is completed, the second step is consuming that media. To that aim, an instance of `RepositoryHttpPlayer` should be created. This object is instantiation from another `RepositoryItem`. Media can be also consumed by means of HTTP, this time using the method GET. Again, this feature enables the integration the NUBOMEDIA Media API, due to the fact that the Media Element `PlayerEndpoint` has been implemented to accept media read from URL by means of the GET method.

4.3.5 API availability

The NUBOMEDIA Repository API is available in the following programming languages

- Java 6.0 or later.

4.3.6 Information for application developers

Once the main features of the API have been introduced, this section is devoted to provide to application developers all the information enabling to create applications using it. This section is designed as a collection high level explanations, pointers and links towards the appropriate developer guidelines and reference information.

Important notice

The NUBOMEDIA Repository API has been implemented as an extension of the Kurento Repository Client API. This means that all documentation, tutorials and knowledge involving the Kurento Repository Client can be directly applied to NUBOMEDIA. Only one aspect needs to be taken into consideration: NUBOMEDIA Media API developers MUST use the kurento-repository-client extended Java SDK, which is used to manage Repository Items through CRUD operations

Tutorials and developer guides

Examples showing how to use the media capabilities exposed trough the Kurento Repository Client can be found here:

- Kurento Repository official documentation
  - http://doc-kurento-repository.readthedocs.org/en/latest/

Reference information

Reference information providing the Javadoc of the Kurento Repository can be found here:

- Kurento Repository Internal JavaDoc.
  - http://doc-kurento-repository.readthedocs.org/en/latest/_static/langdoc/javadoc/internal/index.html
- Kurento Repository Server JavaDoc
  - http://doc-kurento-repository.readthedocs.org/en/latest/_static/langdoc/javadoc/server/index.html
- Kurento Repository Client JavaDoc
  - http://doc-kurento-repository.readthedocs.org/en/latest/_static/langdoc/javadoc/client/index.html

Source code and licensing

The repositories containing the relevant artifacts involved in this API are the following:

- kurento-repository-client
  - https://github.com/Kurento/kurento-java/tree/master/kurento-repository/kurento-repository-client

4.4 The NUBOMEDIA WebRtcPeer API

4.4.1 Objectives

This API has the objective of abstracting client RTC media capabilities, so that its media capture and communication capabilities are exposed to the developer in a simple, seamless and unified way.

4.4.2 Scope

The WebRtcPeer API is specifically concentrated on WebRTC client capabilities. Following W3C WebRTC specifications WebRTC APIs are split into two: the Media Capture API (i.e. getUserMedia) and the PeerConnection API. The former exposes client capabilities for accessing webcam and microphone while the latter enables media communications through an SDP negotiation mechanism. The WebRtcPeer unifies both under a common abstraction.

4.4.3 API Overview and Features

The NUBOMEDIA WebRtcPeer API offers a `WebRtcPeer` object, which is a wrapper of the browser's RTCPeerConnection API. Peer connections can be unidirectional (send or receive only) or bidirectional (send and receive). The following snippet shows how to create the latter in JavaScript, i.e. a `WebRtcPeer` object to send and receive media (audio and video). This code assumes that there are two different video tags in the web page that loads the script. These tags are used to show the video as captured by the browser and the media received from other peer. The constructor receives a property bag that holds all the information needed for the configuration.

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

 var constraints = {
     audio: true,
     video: {
       width: 640,
       framerate: 15
     }
 };

 var options = {
   localVideo: videoInput,
   remoteVideo: videoOutput,
   onicecandidate : onIceCandidate,
   mediaConstraints: constraints
 };

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

      this.generateOffer(onOffer)
   });
```

After executing this code, an `RTCPeerConnection` object is created and then the method `getUserMedia` is invoked. The constraints are used in the invocation, and in this case both microphone and webcam are used. However, this does not create the connection between peers. This is only achieved after completing the SDP negotiation between peers. This process implies exchanging SDPs offer and answer and, since

Trickle ICE is the mechanism carried out by the API. Therefore, a number of candidates describing the capabilities of each peer should be exchanged.

In the previous piece of code, when the `webRtcPeer` object gets created, the SDP offer is generated with `this.generateOffer(onOffer)`. The only argument passed is a function, that will be invoked one the browser's peer connection has generated that offer. The `onOffer` callback method is responsible for sending this offer to the other peer.

Assuming that the SDP offer has been received by the remote peer, it must have generated an SDP answer that should be received in return. This answer must be processed by the `webRtcEndpoint`, in order to fulfill the negotiation. This could be done in the implementation of the `onOffer` callback function.

```
function onOffer(error, sdpOffer) {
  if (error) return onError(error);

  // We've made this function up
  sendOfferToRemotePeer(sdpOffer, function(sdpAnswer) {
    webRtcPeer.processAnswer(sdpAnswer);
  });
}
```

As introduced before, the library assumes the use of Trickle ICE to complete the connection between both peers. In the configuration of the `WebRtcPeer` object, there is a reference to an `onIceCandidate` callback function. The library will use this function to send ICE candidates to the remote peer. In turn, the client application must be able to receive ICE candidates from the remote peer. Assuming the signaling takes care of receiving those candidates, it is enough to invoke the following method in the `webRtcPeer` to gather the ICE candidate:

```
webRtcPeer.addIceCandidate(candidate);
```

4.4.4 Example use cases

There are several ways to use the NUBOMEDIA WebRtcPeer API:

1. By means of the minified JavaScript file. This library can be directly downloaded from the following URL:

   http://builds.kurento.org/release/6.2.0/js/kurento-client.min.js

2. By means of NPM (package manager for Node.js):

   ```
   npm install kurento-utils
   ```

3. By means of Bower (package manager for browser JavaScript):

   ```
   bower install kurento-utils
   ```

There are complete tutorials that show how to use this API in WebRTC applications developed on Java, Node and JavaScript. These tutorials are hosted on GitHub:

- Java: https://github.com/Kurento/kurento-tutorial-java

- Node: https://github.com/Kurento/kurento-tutorial-node
- Javascript: https://github.com/Kurento/kurento-tutorial-js

4.4.5 API availability

The NUBOMEDIA Repository API is currently available in the following languages:

- JavaScript for WebRTC browsers (i.e. Chrome and Firefox on all versions since beginning 2015)
- Java for Android 4.0 and later
- Objective C for iOS on versions 7.0 and later.

These implementations are described in sections below in this document. These implementations are just a wrapper of the `getUserMedia` and `RTCPeerConnection` APIs provided by the WebRTC Chrome stack. Hence, further programming languages might be supported by creating the appropriate wrappers on those APIs.

4.4.6 Information for developers

Documentation

- Browser WebRtcPeer
  - http://doc-kurento.readthedocs.org/en/stable/mastering/kurento_utils_js.html
- Android WebRtcPeer
  - http://webrtcpeer-android.readthedocs.org/en/latest/
- iOS WebRtcPeer
  - https://github.com/nubomediaTI/Kurento-iOS

Source code and licensing

- Browser WebRtcPeer
  - Source: https://github.com/Kurento/kurento-utils-js
  - License: LGPL v2.1
- Androind WebRtcPeer
  - Source: https://github.com/nubomedia-vtt/webrtcpeer-android
  - License: BSD-type license
- iOS WebRtcPeer
  - Source: http://kurento-ios.readthedocs.org/en/latest/index.html
  - License: LGPL v2.1

4.5 The NUBOMEDIA Signaling API

4.5.1 Objectives

This API has been created with the objective of enabling developers to access the NUBOMEDIA JSON-RPC over WebSocket NUBOMEDIA extensible signaling protocol. To this aim, the following sub-objectives have been achieved:

- To provide a signaling API where high level RTC APIs such as the Room API and the Tree API could be created.
- To provide an extensible signaling mechanism for application developers enabling the creation of customized signaling protocols in a seamless manner.

4.5.2 Scope

This API assumes the signaling protocol to be based on JSON-RPC v2 over WebSockets. Hence, it is not appropriate whenever other types of signaling protocols are required. In particular, that signaling protocol enables developers to implement full duplex invocations between clients and servers following a request/response mechanism.

4.5.3 API Overview and Features

This API has two sides: client and server. We introduce them separately. Notice that, for the sake of simplicity, code examples are provided in Java but other programming languages may be supported as specified in the section below.

The NUBOMEDIA Signaling API: message types

Both, server and client side share common data types for representing signaling messages. These types are just a wrapper of the JSON-RPC v2.0 messages, which are the following

Request messages

A RPC call is represented by sending a Request object to a Server. The Request object has the following JSON properties:

- `jsonrpc`: A String specifying the version of the JSON-RPC protocol. In our case, it must be exactly "`2.0`".
- `method`: A String containing the name of the method to be invoked. Method names that begin with the word `rpc` followed by a period character (U+002E or ASCII 46) are reserved for rpc-internal methods and extensions and MUST NOT be used for anything else.
- `params`: A Structured value that holds the parameter values to be used during the invocation of the method. This member may be omitted.
- `id`: An identifier established by the Client that must contain a String, Number, or NULL value if included. If it is not included, the message is assumed to be a notification. The value should normally not be null and numbers should not contain fractional parts. The Server must reply with the same value in the Response object if included. This member is used to correlate the context between the two objects.

Response messages

When a RPC call is made, the Server must reply with a Response, except for in the case of notifications. The response is expressed as a single JSON Object, with the following members:

- `jsonrpc`: A String specifying the version of the JSON-RPC protocol. It must be exactly "`2.0`".
- `result`: This member is required on success. This member must not exist if there was an error invoking the method. The value of this member is determined by the method invoked on the Server.
- `error`: This member is required on error. This member must not exist if there was no error triggered during invocation. The value for this member must be an Object as defined in the section below.
- `id`: This member is required. It must be the same as the value of the id member in the request message. If there was an error in detecting the id in the request message (e.g. Parse error/Invalid Request), it must be Null.

Either the `result` property or `error` property must be included, but both members must never be included.

Error property

When a RPC call encounters an error, the response message must contain the `error` member with a value that is a JSON object with the following members:

- `code`: A Number that indicates the error type that occurred. This must be an integer.
- `message`: A String providing a short description of the error. The message should be limited to a concise single sentence.
- `data`: A Primitive or Structured value that contains additional information about the error. This may be omitted. The value of this member is defined by the Server (e.g. detailed error information, nested errors etc.).

The mapping between the message and the corresponding API data types is language dependent. For example, for Java, request messages are represented through the `org.kurento.jsonrpc.message.Request` class, the response messages through the `org.kurento.jsonrpc.message.Response` class, which in turns contains the error property, which is mapped to the `org.kurento.jsonrpc.message.ResponseError` class. The mapping for other languages is specified with the implementation details provided in sections below.

The NUBOMEDIA Server Signaling API

The NUBOMEDIA Server Signaling API turns around the notion of handlers: developers create custom handlers that provide the logic for handling incoming JSON-RPC requests. Just for illustration, consider the following simple handler

```
public class EchoJsonRpcHandler extends DefaultJsonRpcHandler<JsonObject> {

    @Override
    public void handleRequest(Transaction transaction,
            Request<JsonObject> request) throws Exception {
        if ("echo".equalsIgnoreCase(request.getMethod())) {
            transaction.sendResponse(request.getParams());
        }
    }
}
```

As it can be observed, developers just need to override the `handleRequest` method in order to provide their custom processing logic. In that case, only “echo” messages are processed.

In order to program handlers, developers have some helper classes:

- Transactions: A Transaction represents a request/response pair. Hence, through the transaction developers access the ability of sending back responses to requests.
- Sessions: The Transaction also exposes a Session object. Sessions represent state among different transactions so that attributes can be stored and recovered across them. It is important to remark that this JSON-RPC session concept may spawn across different HTTP (i.e. WebSocket) sessions. This high-level notion of session is enforced through a `sessionId` property which is exchanged

between clients and servers as part of the `params` JSON object, in requests, and of the result JSON object, in responses

The NUBOMEDIA Client Signaling API

The NUBOMEDIA Client Signaling API has the main objective of enabling clients to send JSON-RPC messages to servers and of making possible to receive the corresponding answers and process them. The mechanism for doing so is quite straightforward and can be appreciated in the following code snippet.

```
JsonRpcClient client = new
            JsonRpcClientWebSocket("ws://localhost:8080/echo");

Request <JsonObject> request = new Request<>();
request. setMethod("echo");
JsonObject params = new JsonObject();
params.addProperty("some property", "Some Value");
request.setParams(params);

Response<JsonElement> response = client.sendRequest(request);
```

As it can be observed, once the client object is created pointing to the appropriate WebSocket address, developers can create the requests messages of their choice and ask the client to send them to the server. As a result, a response message is received that can be later analyzed by the client for providing the appropriate processing.

4.5.4 API availability

The NUBOMEDIA server signaling API is available in the following languages:

- Java v6.0 or later

The NUBOMEDIA client signaling API is available in the following languages:

- JavaScript for WWW browsers
- Java for Android 4.0 and later
- Objective C for iOS on versions 7.0 and later.

These implementations are described in sections below in this document. These implementations are just a wrapper of the JSON-RPC v2 over WebSockets protocol. Hence, further programming languages might be supported by creating the appropriate wrappers on those APIs.

4.5.5 Example use cases

The signaling API can be used for establishing any kind of communication between clients and application server logic. Both the NUBOMEDIA room and tree APIs have been created on top of the signaling API and they can be considered as advanced and realistic examples of using it. However, for the objectives of this document, we wish to introduce a very simple example illustrating how the signaling API is used. To this aim, we are going to program the most simple signaling mechanism we can imagine: an echo.

In the echo signaling, clients send text messages to the server which, in turn, answers back the message to the client. Programming an echo signaling with our signaling API is straightforward. First, we need to determine how to create the signaling messages.

This is quite simple given the JSON-RPC message structure specified above: our signaling message just uses the params field and inserts into it the echo string.

After that, we can program the server side logic, which can be implemented with the code shown in the table below:

```
public class EchoJsonRpcHandler extends DefaultJsonRpcHandler<JsonObject> {

  @Override
  public void handleRequest(Transaction transaction,
                            Request<JsonObject> request) throws Exception {

    log.info("Request id:" + request.getId());
    log.info("Request method:" + request.getMethod());
    log.info("Request params:" + request.getParams());

    transaction.sendResponse(request.getParams());

  }

}
```

The client side is also very simple to create, as it can be observed in code snippet below:

```
static class Params {
    String text;
  }

JsonRpcClient client = new JsonRpcClientWebSocket("ws://my.ip.com/path");

    Params params = new Params();
    params.text = "Hello world!";

    Params result = client.sendRequest("echo", params, Params.class);

    LOG.info("Response:" + result);

    log.info(result.text);

    client.close();
```

4.5.6 API availability

The NUBOMEDIA Signaling API is available in the following programming languages

Server side:

- Java v6.0 or latter

Client side:

- JavaScript for browser.
- Android 4.0 or later
- iOS 7.0 or later

4.5.7 Information for application developers

The following information may be interesting for developers wishing to leverage these API capabilities

Documentation

- Server signaling API
  - http://doc-kurento-jsonrpc.readthedocs.org/en/latest/
- JavaScript client signaling API
  - http://doc-kurento-jsonrpc.readthedocs.org/en/latest/
- Android client signaling API
  - http://jsonrpc-ws-android.readthedocs.org/en/latest/
- iOS client signaling API
  - http://kurento-ios.readthedocs.org/en/latest/dev_guide.html#json-rpc

API source code and licensing

The repositories containing the relevant artifacts involved in this API are the following:

- Server signaling API
  - Source
    - https://github.com/Kurento/kurento-java/tree/master/kurento-jsonrpc
  - License
    - LGPL v2.1
- JavaScrpt signaling API
  - Source
    - https://github.com/Kurento/kurento-jsonrpc-js
  - License
    - LGPL v2.1
- Android signaling API:
  - Source:
    - https://github.com/nubomedia-vtt/jsonrpc-ws-android
  - License:
    - BSD-type License
- iOS signaling API:
  - Source:
    - https://github.com/nubomediaTI/Kurento-iOS
  - License:
    - LGPL v2.1

4.6 The NUBOMEDIA Room API

4.6.1 Objectives

This API has the objective of enabling application developers to create group communication applications adapted to real social interactions. To this aim, this objective might be split into the following sub-objectives:

- To design and implement a Room abstraction suitable for exposing to application developers the ability of managing group communications.
- To design and implement a mechanism enabling developers to publish users' media streams to the room whenever appropriate.
- To design and implement a mechanism enabling developers to subscribe users to media publishing events in the room, so that they can receive others' streams whenever appropriate.
- To make possible to comply with all the above specified objectives in a PaaS cloud environment.

4.6.2 Scope

The NUBOMEDIA Room API can be used safely as long as the following requirements are satisfied:

- The API assumes that all clients participating in a room are based on WebRTC transports. The API does not provide support for other types of RTC transports off-the-shelf.
- The API is based on the NUBOMEDIA Signaling Protocol (i.e. a custom protocol based on JSON-RPC over WebSocket). Hence, the API cannot interoperate with other types of signaling mechanisms such as SIP or XMPP.

4.6.3 API Overview and features

The NUBOMEDIA Room API has been designed for the development of real time conferencing applications basin on room models. In these, each group of participants share a virtual space known as "room" where different resources (e.g. media streams, chat messages, etc.) are shared among the members but isolated to members of other group. The room API makes possible to manage rooms and participants as well as the communication resources they require in a Kurento Media Server instance. The architecture of an application based on the NUBOMEDIA Room API is illustrated in Figure 8 and comprises the following modules (from top to bottom):

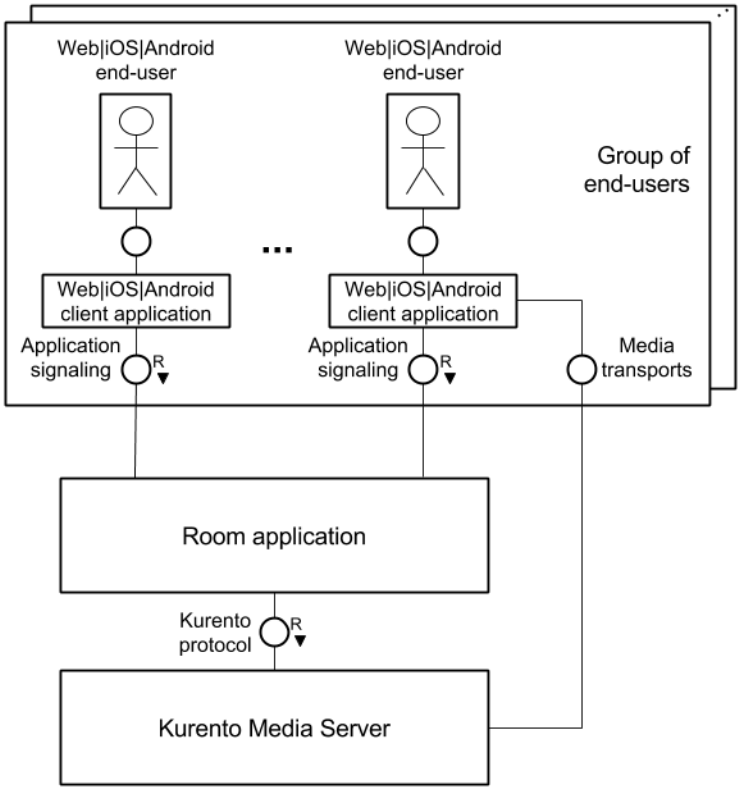


Figure 8. Architecture of a Room application

- Client application: The client application (Web|iOS|Android client applications as shown in Figure 8) provides a GUI for the room subscribing to different relevant events in the room (e.g. new stream available, new user arrived) and to execute actions upon their reception (e.g. receive a media stream; remove a user from the client GUI, etc.)

- Application signaling: It is a protocol based on the NUBOMEDIA signaling mechanisms. This protocol is in charge of the signaling exchanges between the client and the application server.
- Room application: The room application manages the logical notion of room and participant translating the different requests received from applications into the appropriate control messages into the media server so that the media capabilities required by the room are in place.

This room application architecture has been made adaptable and customizable so that both the client application and the room application can implement custom logic adapted to developers' needs. For this, we have created a number of APIs that make possible to create such type of room architectures in a simple and seamless way. These include the Room Client API, the Room Protocol and the Room Server API. The interaction of these components among each other and their relationships with the client and application server room logic is depicted in Figure 9.

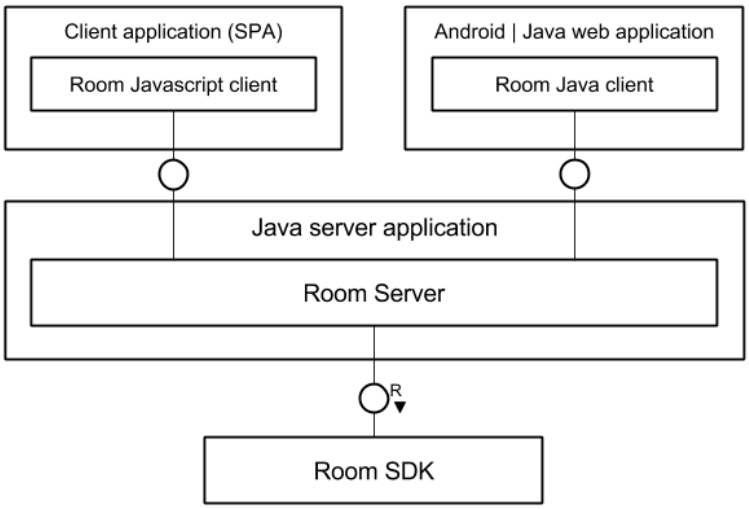


Figure 9. Integration of the Room API components

The NUBOMEDIA room client API has several flavors (Java, JavaScript, iOS, etc.) It is in charge of exposing to application developers the appropriate capabilities for creating room GUIs. Intuitively, the room client API makes possible for the client application logic to subscribe to relevant events happening on the room in relation to other users (e.g. new user entered the room, user left the room, etc.) and to send commands to the server side logic in relation to the user owning the room (e.g. join the room, leave the room, publish media to the room, etc.)

The NUBOMEDIA room server API (called room SDK in the picture for simplicity) is in charge of managing to main logical entities:

- `rooms`: It represents groups of peers that are able to communicate each other by means of WebRTC. Each user can be part of only one room at a time.
- `participants`: Users within the room. The application must provide an identifier of the user (parameter `participantId`).

The room server API receives commands from the room client API through the room protocol and provides semantics to them. These semantics is either associated with room or participants lifecycle (i.e. creation or deletion) either related to the media

exchanges that take place among participants in a room. This semantics requires many times to instantiate and manipulate low level media capabilities held my a media server

4.6.3.1 The Room Client API

The Room Client API is built around three main classes: `Room`, `Participant` and `Stream`.

The `Room` class is an abstraction that provides access to local and remote participants and their streams. For this, it exposes a number of primitives and events, which are the following:

Primitives

- `connect()`: makes possible for a user to connect to the room.

Events

- `room-connected`: indicates a successful connection to the room. This event is published together with the list of participants currently in the room and with the list of media streams available in the room.
- `error-room`: indicates an error in the room as well as a description of the error.
- `room-closed`: indicates that a room was terminated.
- `participant-joined`: notifies about a new participant in the room.
- `participant-left`: notifies about a participant leaving the room.
- `participant-evicted`: indicates that a participant has been evicted from the room.
- `stream-published`: notifies about the availability of a new media stream.
- `stream-subscribed`: notifies that a subscribe operation to a stream has been successful.
- `stream-added`: notifies that the room automatically added and subscribed to a published stream.
- `stream-removed`: indicates that a stream is no longer available typically due to a participant disconnecting from the room.
- `newMessage`: indicates a new text message has been published into the room. This is useful for implementing chat applications.

`Participant`: represents a specific participant (or peer) which can be local or remote. It is a data structure holding all the relevant information about a specific participant.

`Stream`: is a wrapper class for manipulating the media streams published in the room. It has the following primitives and events:

Primitives

- `init()`: it is used only for local streams. It triggers a request towards the user to grant access to the local camera and microphone.

Events

- `access-accepted`: emitted when the user grants access to the camera and microphone.
- `access-denied`: emitted when the user does not grant such access.

4.6.3.2 The Room Server API

The Room server API is architected following the scheme depicted in Figure 10.

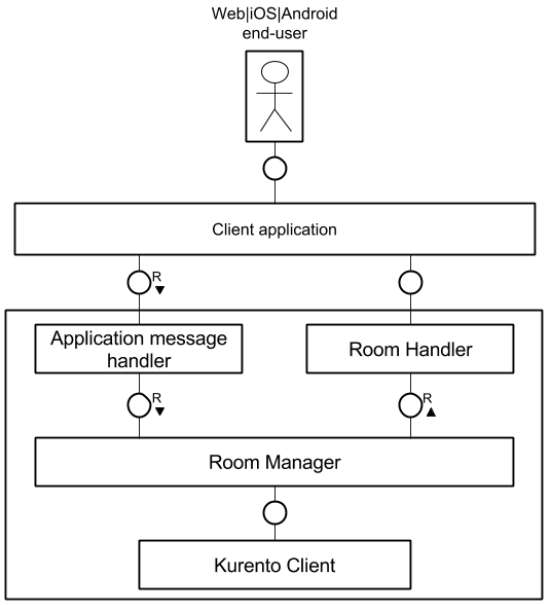


Figure 10. Architecture of the Room Server API

The Client application and the application server logic exchange room signaling messages through the room protocol. These messages are manage by the application message handler, which is just a handler of the NUBOMEDIA signaling API as specified in sections above. This handler then is in charge of translating such signaling messages into the appropriate invocations into the Room Manager. As it can be observed, developers can include any kind of custom logic for performing such translation including specific AAA (Authentication Authorization and Accounting) policies, specific media processing, etc.

The Room Manager is the main class of the room server API. It is in charge of managing the room and participant abstractions and on providing the semantics for the room protocol messages. Among other, the Room Manager exposes to the application developer primitives such as the following:

- `joinRoom`: used for requesting a new participant to join the room.
- `leaveRoom`: removes a participant from the room.
- `publishMedia`: negotiates a WebRTC media exchange with the room.
- `unpublishMedia`: removes a previously published WebRTC stream from the room.
- `onIceCandidate`: notifies of a new ice candidate coming from the client in a WebRTC session establishment.
- `sendMessage`: publish an instant message to the room.

As a consequence of the semantics of Room Manager invocations asynchronous events shall emerge coming both from the media server and from the room management logic. These events are notified back to the clients through a Room Handler, which handles room events. The room API provides a default Room Hander which just send back the corresponding signaling message to the clients. However, developers can override it for implementing customized business logic. The Room Handler implements, among others, the following handling primitives:

- `onParticipantJoined`: invoked when a new participant joins the room.

- `onParticipantLeft`: invoked whenever a participant leaves the room.
- `onPushlishMedia`: invoked when a new media stream is published to the room.
- `onUnpublishMedia`: invoked when a media stream is unpublished.
- `onSendMessages`: invoked when a text message is send to the room

4.6.4 Example use cases

The typical example of an application using the Room API is Single Page Application (SPA) based on the Room Server and the Room JavaScript Client. This application enables users to simultaneously establish multiple connections to other users connected to the same session or room. These steps to implement this application are the following:

- Include the SDK module to the dependencies list.
- Create one `RoomManager` instance by providing an implementation for one of the following interfaces: `RoomEventHandler` or `KurentoClientProvider`.
- Develop the client-side of the application for devices that support WebRTC (or use `client-js` library and take a look at the demo's client implementation).
- Design a room signaling protocol that will be used between the clients and the server (or use the WebSockets API from `room-server`).
- Implement a handler for clients' requests on the server-side that use the `RoomManager` to process these requests (hint: JSON-RPC handler from `room-server`).
- Choose a response and notification mechanism for the communication with the clients (JSON-RPC notification service from `room-server`).

4.6.5 API availability

The NUBOMEDIA Room API is available in the following programming languages

Server API

- Java 6.0 or later

Client API

- JavaScript
- Android 4.0 or later
- iOS 7.0 or later

4.6.6 Information for developers

Once the main features of the API have been introduced, this section is devoted to provide to application developers all the information enabling to create applications using it. This section is designed as a collection high level explanations, pointers and links towards the appropriate developer guidelines and reference information.

Important notice

The NUBOMEDIA Room API has been implemented as an extension of the Kurento Room API. This means that all documentation, tutorials and knowledge involving the Kurento Room API can be directly applied to NUBOMEDIA. Only one aspect needs to be taken into consideration: NUBOMEDIA Room API developers MUST use the kurento-client extended Java SDK.

Documentation
The following references provide links to the official documentation which includes tutorials, examples and reference information:

- Room server API official documentation
  - http://doc-kurento-room.readthedocs.org/en/latest/
- Room protocol API official documentation
  - http://doc-kurento-room.readthedocs.org/en/latest/websocket_api_room_server.html
- WWW room client API official documentation
  - http://doc-kurento-room.readthedocs.org/en/latest/client_javascript_api.html
- Android room client API official documentation
  - http://kurento-room-client-android.readthedocs.org/en/latest/
- iOS room client API official documentation
  - http://kurento-ios.readthedocs.org/en/latest/dev_guide.html#kurento-room

Source code and licensing
The repositories containing the relevant artifacts involved in this API are the following:

- Room server API
  - Source:
    - https://github.com/Kurento/kurento-room/tree/master/kurento-room-sdk
  - License:
    - LGPL v2.1
- Room Javascript client API
  - Source:
    - https://github.com/Kurento/kurento-room/tree/master/kurento-room-client-js
  - License:
    - LGPL v2.1
- Room Android client API
  - Source: https://github.com/nubomedia-vtt/kurento-room-client-android
    -
  - License:
    - LPGL v2.1
- Room iOS client API
  - Source:
    - https://github.com/nubomediaTI/Kurento-iOS
  - License:
    - LGPL v2.1

4.7 The NUBOMEDIA Tree API

4.7.1 Objectives

The NUBOMEDIA Tree API allows developers to build video broadcasting web applications. It is developed using WebRTC technology on the top of Kurento Media Server. The broadcasting model is based in the Tree Topology. This model allows distributing WebRTC media from a presenter to a large number of viewers in a simple way.

The main aim of the Tree API is scalability. Instead of one-to-many WebRTC communications, the tree API has been designed to broadcast media using different instances of Kurento Media Server. The idea is connecting a WebRTC source to different viewers (sinks), and when an instance of Media Server is over loaded, other instances of Media Server can be used to distribute media. Figure 11 illustrates this concept. In this picture, we can see an especial endpoint designed in the context of NUBOMEDIA (labeled as "WebRtc Conn.", i,e. `WebRtcEndpoint` connector). This element has the capability of connecting media from different pipelines.

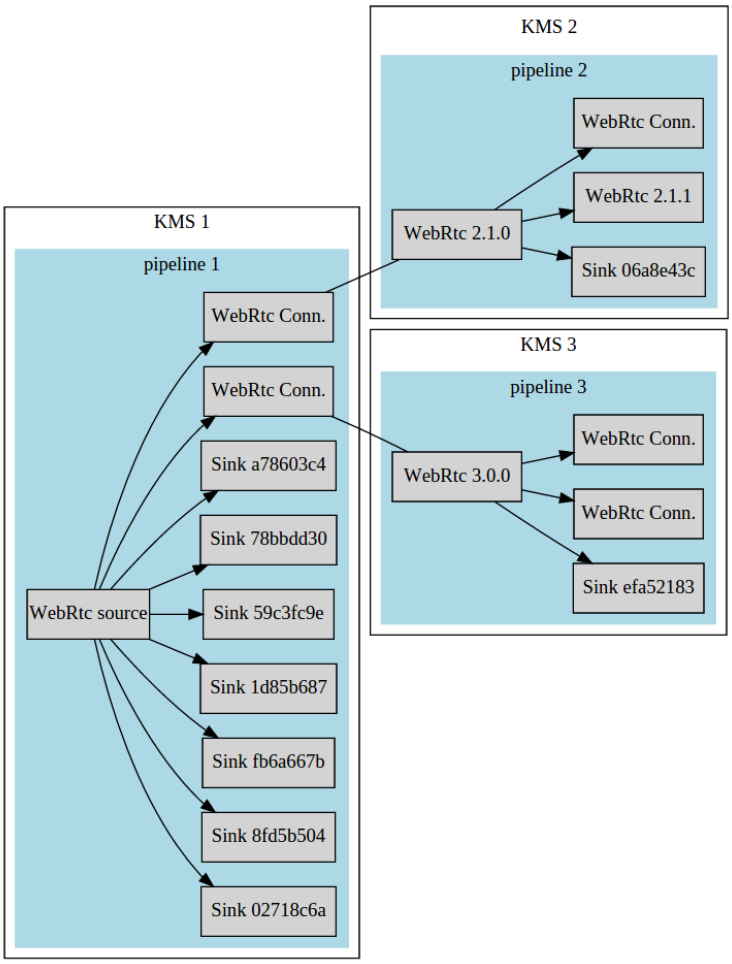


Figure 11. Example of Tree API Scalability

4.7.2 Scope

The NUBOMEDIA Tree API has the following requirements:

- The API assumes that all clients participating in a Tree are based on WebRTC transports.
- The Tree API is based on the NUBOMEDIA Signaling Protocol (i.e. a custom protocol based on JSON-RPC over WebSocket).

4.7.3 API Overview and Features

The NUBOMEDIA Tree API is based on client-server architecture. Concretely, it is composed by a server-side library and two client-side libraries (Java and a JavaScript):

- `tree-server` is a component of the API designed to be deployed and controlled by clients. This component uses a Kurento Media Server instance to provide WebRTC media handling to clients.

- `tree-client` implements a Tree Client designed to be used in Java web applications or Android applications. The client contains an implementation of JSON-RPC WebSocket protocol implemented by Tree server. It does not contain any functionality to control video players or video capturing from webcams.
- `tree-client-js` implements a Tree Client to be used in Single Page Applications (SPA). Besides the JSON-RPC WebSocket protocol, it uses several libraries to control WebRTC and HTML5 APIs in the browsers. This allows to developer to focus in its application functionality hiding low level details.

The following picture shows a high-level architecture of these components. In this chart, two different clients connect to the server by means of JSON-RPC over WebSocket. The TreeServer is charge of distributing media using one or several instances of Media Server. In order to use different instances of Media Server, the kurento-client extended SDK can be used.

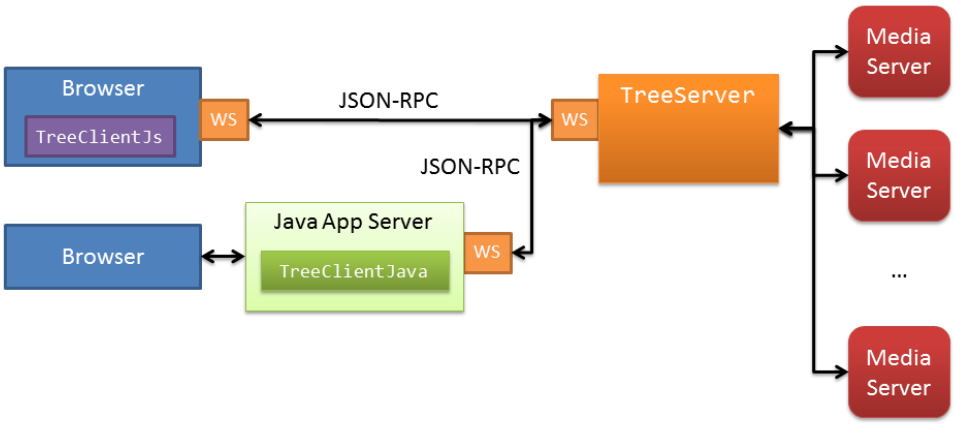


Figure 12. Tree Overview

4.7.3.1 The Tree Client API

The Tree Client API is built in one main class: `KurentoTreeClient` (named just `KurentoTree` in the JavaScript client). This class provides and abstraction of the Tree Topology, allowing clients to feed/consume WebRTC media from/to the media broadcasting service. It exposes the following primitives:

- `createTree()`: This operation allows creating a WebRTC broadcasting tree. If no parameters are provided, it returns a unique Tree identifier (`TreeId`). It also accepts to handle this identifier manually (i.e., passing the `TreeId` value as parameter).
- `setTreeSource()`: This operation allows to establish the presenter (source) in the Tree. To that aim, the Tree should be previously created and its identifier (`TreeId`) should be available. In order to carry out the WebRTC media negotiation, the SDP message should be passed as parameter in this operation. It returns an identifier for the source (`SourceId`).
- `removeTreeSource()`: This operation is called when the presenter is not the source of the media anymore.
- `addTreeSink()`: Once a source is available in the Tree, the next step is consume that media. This is done by adding sinks (i.e. viewers) to the Tree. This primitive implements this feature. Again, the `TreeId` is used to identify the Tree, and the SDP of the sink is required to complete the WebRTC media negotiation. This operation returns an identifier for the source (`SinkId`).

- `removeTreeSink()`: Sinks can be dropped from the Tree with this method. The parameter `SinkId` is required to identify the sink to be erased.
- `addIceCandidate()`: Besides the SDP negotiation, it is necessary to exchange ICE candidates between client and server (Trickle ICE mechanism). This operation is called to send an ICE candidate from the client the server.
- `getServerCandidate()`: This operation polls the candidates list maintained by this client to obtain a candidate gathered on the server side.
- `releaseTree()`: This operation is called to destroy a Tree previously created (identified by `TreeId`).

4.7.3.2 The Tree Server API

The Tree server exposes a SockJS[1] WebSocket at http://treeserver:port/kurento-tree, where the hostname and port depend on the current setup. The exchanged messages between server and clients are JSON-RPC 2.0 requests and responses. Other clients than Java/JavaScript can be implemented if follow the JSON-RPC over WebSocket protocol. The events are sent from the server to client as notifications (they don't require a response and they don't include an identifier). The following table summarizes the possible operations provided by the Tree service:

Operation	Description
Create tree	Request to create a new tree in Tree server. It is send by clients to server.
Set tree source	Request to configure the emitter (source) in a broadcast session (tree). It is send by clients to server.
Remove tree source	Request to remove the current emitter of a tree. It is send by clients to server.
Add tree sink	Request to add a new viewer (sink) to the tree. It is send by clients to server.
Remove tree sink	Request to remove a previously connected sink (viewer). It is send by clients to server.
Ice candidate	Notification sent form server to client when a new Ice candidate is received from Kurento Media Server. It is send by server to clients.
Add ice candidate	Request used to add a new ice candidate generated in the browser. It is send by clients to server.
Remove tree	Request used to remove a tree. It is send by clients to server.

Tree Server has been implemented in Java. The core class in this server is the `TreeManager`. This entity is in charge of handling the requests from clients. It implements a simple algorithm to place Tree sinks (viewers) in any Media Server instance. Figure 13 shows the class diagram of `TreeManager`. In this picture we can see another important entity: `ClientJsonRpcHandler`. This class is in charge of implementing the JSON-RPC over WebSocket communication with clients.

[1] https://github.com/sockjs

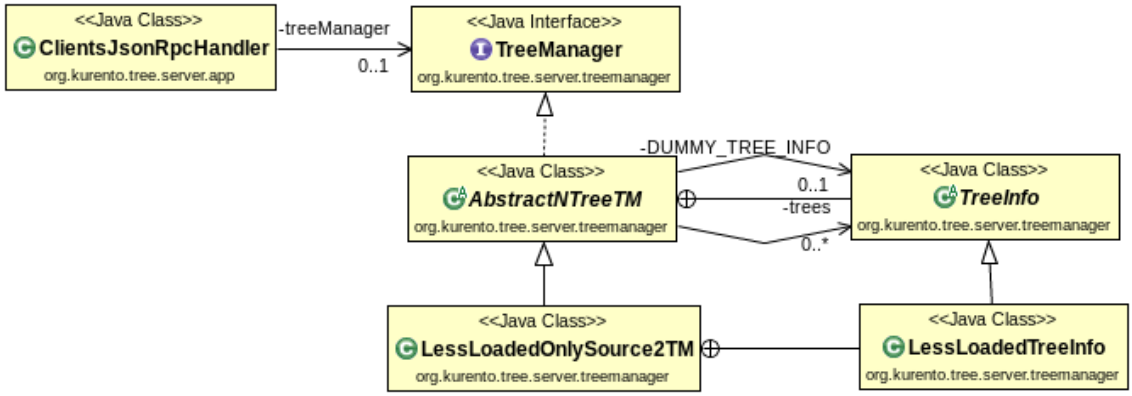


Figure 13. `TreeManager` class diagram

4.7.4 Example use cases

The typical scenario of an application using the NUBOMEDIA Tree API is a web application that allows a user to broadcast his webcam and any other user can see it.

From a technical point of view, this use case can be implemented as a Single Page Application (SPA) as a bunch of HTML, CSS and JS files. These files are served from a SpringBoot web server, but can be served with any http server. The JavaScript code uses Tree JavaScript client to communicate directly with Tree Server.

JavaScript logic communicates with SpringBoot server by means of WebSocket protocol. The SpringBoot server handle WebSocket messages and uses Tree Client to communicate with Tree Server. That is, there is no direct communication between JavaScript code and Tree Server. All communications is through SpringBoot server app.

4.7.5 API availability

The NUBOMEDIA Tree API is available in the following programming languages

Server API

- Java 7.0 or later

Client API

- Java
- JavaScript

4.7.6 Information for application developers

This section contains information aimed to guide developers using the NUBOMEDIA Tree API.

Important notice

The NUBOMEDIA Tree API has been implemented as an extension of the Kurento Tree API. This means that all documentation, tutorials and knowledge involving the Kurento Tree API can be directly applied to NUBOMEDIA. Only one aspect needs to be taken into consideration: NUBOMEDIA Tree API developers MUST use the kurento-client extended Java SDK.

Documentation

The following references provide links to the official documentation which includes tutorials, examples and reference information:

- Tree API official documentation
  - http://doc-kurento-tree.readthedocs.org/en/latest/
- Tree server protocol API official documentation
  - http://doc-kurento-tree.readthedocs.org/en/latest/deployment.html
- Tree JavaScript client API official documentation
  - http://doc-kurento-tree.readthedocs.org/en/latest/javascript_client.html
- Tree Java client API official documentation
  - http://doc-kurento-tree.readthedocs.org/en/latest/java_client.html

Source code and licensing

The repositories containing the relevant artifacts involved in this API are the following:

- Tree server and Java/JavaScript clients:
  - Source:
    - https://github.com/Kurento/kurento-tree
  - License:
    - LGPL v2.1

5 NUBOMEDIA APIs implementation

5.1 NUBOMEDIA Media API implementation

5.1.1 The NUBOMEDIA Media API IDL

We designed the NUBOMEDIA Media API to be programming language agnostic so that it could be available for wider development audiences. Due to this, the NUBOMEDIA Media API capabilities are specified through an IDL (Interface Definition Language), which does not depend on any specific programming language. From an implementation perspective that IDL is compiled later to different in order to generate the appropriate SDKs. In this way, NUBOMEDIA Media API capabilities are defined only once but the corresponding implementations can be generated for a variety of languages.

For simplicity, we have decided the NUBOMEDIA Media API IDL to be based on a JSON notation. In an NUBOMEDIA Media API file there are four sections: `remoteClasses`, `complexTypes`, `events` and `code`:

- The `remoteClasses` section is used to define the interface to media server objects. We call it "remote" because these objects are remote from the perspective of the API consumer, as they are hosted into the RTC media server. For example, `PlayerEndpoint` and `ImageOverlayFilter` are defined in this section in their corresponding IDL file.
- The `complexTypes` section is used to define enumerated types and registers used by remote classes or events. For example, the enumerated type `MediaType` with possible values AUDIO, DATA or VIDEO may be defined in this section.
- The events section is used to define the events that can be fired when using the NUBOMEDIA Media API. For example, `EndOfStream` may be defined in the events section of the IDL file describing a `PlayerEndpoint`, so that the event if fired when the end of the stream is reached by the player.

- The code section is used to define properties to control the code generation phase for different programming languages. For example, in this section we can specify the package name in which all artifacts are generated for Java language.

The code snippet shown below outlines an example of an IDL file. For the sake of simplicity, we have replaced with dots (...) some parts it.

```
{
  "code": {
     "api": {
         "java": {
             "packageName": "org.kurento.client",
             ...
         }
     }
  },
  "remoteClasses": [
     {
        "name": "PlayerEndpoint",
        "extends": "UriEndpoint",
        "constructor": {
           "params": [
              { "name": "mediaPipeline", "type": "MediaPipeline" },
              { "name": "uri", "type": "String" }
           ]
        },
        "properties": [
           { "name": "position", "type": "int64" }
        ],
        "methods": [
           { "name": "play", "params": [] }
        ],
        "events": [ "EndOfStream" ]
     },
     ...
  ],
  "events": [
     {
        "name": "EndOfStream",
        "extends": "Media",
        "properties": []
     },
     ...
  ],
  "complexTypes": [
     {
        "name": "MediaType",
        "typeFormat": "ENUM",
        "values": [ "AUDIO", "DATA", "VIDEO" ]
     },
     {
        "name": "Fraction",
        "typeFormat": "REGISTER",
        "properties": [
           { "name": "numerator", "type": "int" },
           { "name": "denominator", "type": "int" }
        ]
     },
     ...
```

```
  ],
  ...
}
```

As it can be observed, to define a remote class in Media API IDL it is mandatory to assign it a name. In addition, the following fields can be incorporated:

- **Extends:** A remote class may extend another remote class. In this case, all properties, methods and events of the superclass are available in objects of the subclass. Note that constructors of the superclass are not inherited. That is, they cannot be used to create objects of the subclass.
- **Constructor:** A remote class constructor is defined with a parameter list. Every parameter has a name and a type. The available types are: primitive types (`String`, `boolean`, `float`, `double`, `int` and `int64`), remote classes or complex types. Parameters can be defined as optional.
- **Properties:** A property is a value associated to a name. To define a remote class property it is necessary to specify its name and type. Properties can be defined as "read only".
- **Methods:** Methods are named procedures that can be invoked with or without parameters. Every parameter is specified by its name and type. Parameters can be defined as optional. A return type can be specified if the method returns a value.
- **Events:** If a remote class declares an event it means that events of this type can be fired by objects of this remote class. It depends on the target programming language how these events are processed.

Remote classes are used mainly to define the `MediaElements` of the NUBOMEDIA Media API. To define a new `MediaElement` the only requirement is define a new remote class that extends the built-in `MediaElement` remote class. This super class defines the properties, methods and events of all `MediaElements`. The `MediaElement` class extends the `MediaObject` class, creating the class hierarchy represented in Figure 14.

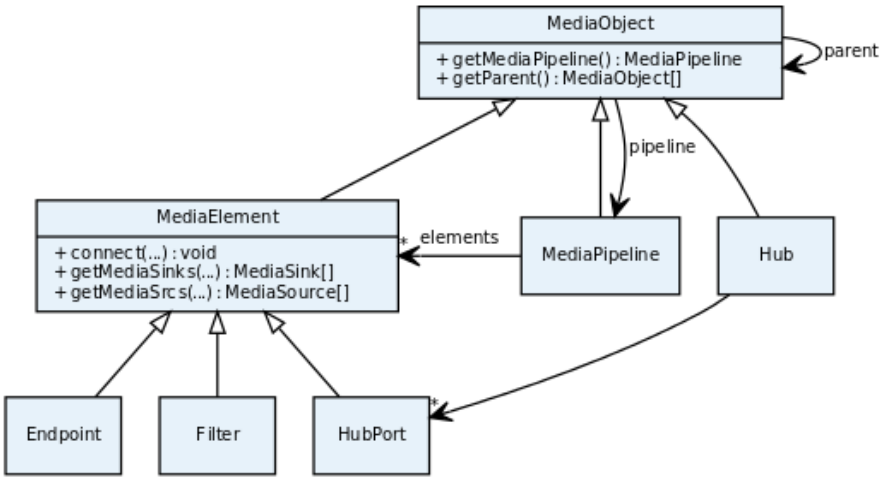


Figure 14. MediaObject UML (Unified Modeling Language) inheritance diagram as defined in the NUBOMEDIA Media API IDL specification

To define an event, it is mandatory to assign it a name. In addition, an event can have properties. Every property must be defined with name and type. In the same way than remote classes, events can also extend a parent event type inheriting all its properties.

Regarding complex types, they can have two formats: `enumerated` or `register`. If a property or `param` is defined with an enumerated complex type, it can only hold a value from the list of specified values. For example, properties based on the enumerated complex type `MediaType` shown on code snippets above must have the value AUDIO, DATA or VIDEO. On the other hand, register complex types can hold objects with several properties. For example, the register complex type Fraction has two int properties: numerator and denominator.

To conclude, the code section is used to specify language-dependent configurations to the IDL compiler. Every programming language has its own section to avoid collisions. For example, the Java package name of the generated code has only sense in Java, while the name of the node module has only sense in JavaScript.

5.1.2 Compiling the NUBOMEDIA Media API IDL.

The IDL format described above makes possible to define the NUBOMEDIA Media API modules in a language-agnostic way. However, this needs to be translated into programming-language-dependent interfaces in order to have the real APIs to be used by application developers. The IDL compiler performs that task. Hence, we need to specify how this compilation happens so that all compiler implementations maintain compatibility on the generated code. For illustration, we have created such specification and as well as the compilers for the two most popular programming languages in the WWW: Java and JavaScript. At the time of this writing, only the Java implementation is fully supported by NUBOMEDIA, being the JavaScript implementation only suitable for working directly with Kurento Media Server. However, for the sake of completeness, we include the implementation details for both compilers.

The Java IDL compiler works in the following way:

- Package: all artifacts (i.e. classes, interfaces and enums) are generated in the package specified in `code.api.java.packageName` section of JSON IDL file.
- Remote classes: For every remote class there are two generated artifacts: an interface and a builder class:
    - Interface: For every remote class a Java interface is generated. This interface has the remote class methods defined in the IDL. In addition, for every property, a getter method is also included. The name of the method is the string "`get`" followed by property name. If the property is not read only, also a setter method is generated following the same approach. Finally, for every event declared in the remote class, a method to subscribe listeners to it is generated. For example, the PlayerEndpoint has the event `EndOfStream` declared in the IDL so the method String `addEndOfStreamListener(Listener<EndOfStream> listener)` is generated. The complementary method to remove the subscription is also generated. `Listener<E>` is a generic interface with only one method: `onEvent(E event).`
    - Builder class: We use builder classes to create a new remote class instances. A Builder is generated for each remote class. All mandatory params in the remote class constructor are mapped to parameters to the only constructor of the builder class. In this way, the compiler enforces that all mandatory parameters have a value. Optional constructor parameters are generated in builder class as fluent setter methods (prefixed with "with" instead of "set" or not prefixed if the method starts with "use"). The builder class is generated as an internal type of the

above-mentioned interface to associate easily the class and the interface. The code snippet below shows the creation of a PlayerEndpoint with the optional constructor parameter `useEncodedMedia` set to true.

```
PlayerEndpoint player = new PlayerEndpoint.Builder(pipeline,
    "video.webm").useEncodedMedia().build();
```

- Complex types: Depending on the complex type format (enum or register) the code generation is different:
  - Enumerated complex type: A Java enum class is generated.
  - Register complex type: A basic Java bean class is created. For every property, a getter and setter method is generated. In addition, a constructor with all properties as parameters is also generated. The code snippet below shows a sample code using a register (`WindowParam`) as a constructor parameter of a `PointerDetectorFilter` remote class.

```
PointerDetectorFilter pointer = new PointerDetectorFilter.Builder(
     pipeline, new WindowParam(5, 5, 30, 30)).build();
```

- Events: For each event defined in a NUBOMEDIA Media API IDL file a new Java class is generated. The name of the class is suffixed with "`Event`". This class is very similar to the generated classes for register complex types. That is, a getter and a setter method is included for every property. In addition, all event classes extends from the `RaiseBaseEvent` base class. This base class contains properties for holding the source of the event (`source`) and the timestamp in which the event was generated (`timestamp`). The code snipped below shows an example illustrating how to work with events.

```
PlayerEndpoint player = new PlayerEndpoint.Builder(pipeline,
  "video.webm").useEncodedMedia().build();

//Java 7
player.addEndOfStreamListener(new EventListener<EndOfStreamEvent>()
{
  public void onEvent(EndOfStreamEvent e) {
     System.out.print("EOS Player "+e.getSource().getId());
  }
});

//Java 8
player.addEndOfStreamListener(
  e -> System.out.print("EOS Player "+e.getSource().getId())
);
```

When working with JavaScript IDL compilers, equivalent rules have been created:

- Package: We base on the NPM (Node Package Manager) JavaScript packaging system. Following this, a package.json file is generated. The following values are used:
  - package name: `code.api.js.nodeName`
  - package description: `code.api.js.npmDescription`
- Remote classes: For every class a new JavaScript prototype based class is generated. This class has all methods defined in the IDL file. In addition, for

every property, a getter method is generated. Also setter methods are generated for non-read only properties. All generated methods have the parameters defined in the IDL plus a callback function. That callback parameter is used to implement the async execution of the method given that the API primitive may require communicating with the RTC media server and, hence, cannot be synchronous. To create an object from a remote class, a factory method called create and available at the pipeline object needs to be executed. The first parameter of the method is the name of the remote class to create as a string. The second is an options bag used when constructor parameters are required. The third, and last, is the async callback to receive the new object handler or an error. The code snippet below shows the creation of a `PlayerEndpoint` with the mandatory parameter uri and optional constructor parameter `useEncodedMedia` set to true. As it can be observed, media element creation is an async operation.

```
pipeline.create("PlayerEndpoint",
                {uri:"video.webm",useEncodedMedia:true},
                function(error,player) {
                  if (error) return console.error(error);

                 //use player here
                });
```

- Complex types: For enumerated complex types, there is no code generation. Enum values are simply strings. On the other hand, register complex type are generated as JavaScript classes based on prototypes. Also, for every register complex type a factory function is generated to allow the creation of objects. The following code snipped shows the creation of a `PointerDetectorFilter` using a complex type `WindowParam` as parameter.

```
var options = {
 "windowParam":WindowParam({
    "topRightCornerX":5,
    "topRightCornerY":5,
       "height":30,
    "width":30
  })
};
pipeline.create("PointerDetectorFilter",
                options,
                function(error, pointer) {
                   if (error) return console.error(error);

                });
```

- Events: There are no classes generated for events in JavaScript. When an event is raised, a new object is created and populated with all relevant information as properties. In the following piece of code, a Player is created and a listener is registered for its event "`EndOfStream`". When this event is generated, a function is executed with the event as parameter. This event parameter can be used to obtain the relevant information such us timestamp, source of the event, etc.

```
pipeline.create("PlayerEndpoint",
                {uri:"video.webm",useEncodedMedia:true},
                function(error, player) {
                     if (error) return console.error(error);
                     player.on("EndOfStream", function(e){
                        console.log("EOS player "+e.source.id)
                     });
                });
```

5.1.3 Creation and deletion of media capabilities

Java and JavaScript have notable differences in media object creation. This is due to the differences in the type safety of both languages. Java is strongly typed. Hence, it is important that the compiler enforces typing in several contexts: mandatory parameters, optional parameters, media object signature, etc. On the other hand, in JavaScript there is no type checking until runtime and this is why we do not enforce any kind of protection.

The releasing of media objects is simpler. We consider that a media object is released when the release method is invoked. In Java, the release method can be executed in a synchronous way, blocking the invoking thread until response is received. That response can be successful or fail. In the latter case, an exception is thrown. In JavaScript, it is executed asynchronously. For this reason, a callback parameter is necessary so that failures can be notified. The following piece of code shows the release of a media object in Java and in JavaScript.

```
//Media object release in JavaScript
player.release(function(error){
  if (error) return console.error(error);
});

//Media object release in Java
player.release();
```

5.1.4 Synchronous and asynchronous programming models

One of the most critical design decisions when designing APIs is how they behave in relation to threads. When performing I/O (Input/output) operations, there is a common agreement that asynchronous APIs are more scalable than synchronous ones. Synchronous I/O typically block threads until response is received or a timeout is reached. Hence, given that there is a practical limit on the number of threads in a system (mainly due to memory constraints), synchronous API models tend to generate thread starvation and decrease performance due to the overload they generate into the operating system task scheduler. To solve this problem, many modern APIs provide asynchronous I/O operations. In this case, the thread executing the I/O is not blocked after the invocation and can be used to execute other tasks. However, asynchronous APIs are more complex to use and are susceptible of suffering a problem called "callback hell". This is a well-known problem that arises when asynchronous calls are invoked in the callbacks of other asynchronous calls, creating a deep nesting of callbacks.

When we designed the NUBOMEDIA Media API we decided to provide developers the flexibility of choosing between the synchronous and the asynchronous models so that

they do not fell limited by any of their corresponding drawbacks. Due to this, our Java IDL compiler generates two methods of each I/O operation: one synchronous and the other asynchronous. Synchronous methods block the calling thread until a response is received. This can be appreciated, for example, in the code snippet shown below. After that, the execution continues. The asynchronous primitives, in turn, include as a last parameter a continuation, that is, an object that have two methods: `onSuccess`, that is executed when the response is received, and `onError`, that is executed when an error or timeout occurs. The code snippet below shows an example.

```
new PlayerEndpoint.Builder(pipeline, "video.webm")
    .buildAsync(new Continuation<PlayerEndpoint>() {

      @Override
      public void onSuccess(PlayerEndpoint player) throws Exception {

        player.play(new Continuation<Void>() {

          @Override
          public void onSuccess(Void result) throws Exception {
             log.info("Play started");
          }

          @Override
          public void onError(Throwable error) throws Exception {
            log.error("Exception invoking play in player", error);
          }
        });
      }

      @Override
      public void onError(Throwable error) throws Exception {
        log.error("Exception creating player", error);
      }
    });
```

When going to JavaScript, things are more complex. Due to the characteristics of the JavaScript language both in the browser and in Node.js, only asynchronous I/O operations are possible. Due to this, and as it can be seen in the code snippets shown below, our IDL compiler includes a callback as the last parameter that is executed asynchronously when the operation is resolved. However, providing only this mechanism reduces the flexibility of developers to avoid the callback hell. Due to this, we designed novel mechanisms for simplifying developers' work. The first one is based on Promises. A Promise represents an operation that has not completed yet, but is expected to do so in the future. Hence, an asynchronous method can return a promise object instead of expect a callback as last parameter. The developer specifies the code to be executed when the promise is fulfilled, executing a method called "then" with the callback as parameter. As the code below shows a JavaScript code creating a player and invoking the play method on it comparing the traditional implementation based on callbacks with an implementation using promises.

```
//Asynchronous API with callbacks
pipeline.create("PlayerEndpoint",{uri:"video.webm",useEncodedMedia:true},
 function(error,                          player)                          {
    if (error) return console.error(error);
```

```
    player.play(function(error){
      if (error) return console.error(error);
      console.log("Play started");
    });
});

//Asynchronous API with promises
pipeline.create("PlayerEndpoint",{uri:"video.webm",useEncodedMedia:true})
 .then(function(player){ return player.play(); })
 .then(function(){ console.log("Play started"); })
 .catch(function(err){ console.error(error); });

//Asynchronous API with promises and ES6 arrow functions
pipeline.create("PlayerEndpoint",{uri:"video.webm",useEncodedMedia:true})
 .then(player => player.play())
 .then(() => console.log("Play started"))
 .catch(err => console.error(error));
```

Moreover, if promises are combined with generators, a new ES6 (ECMAScript 6) feature, the asynchronous code can look like synchronous one. For illustration, remark that code snippet below implements the same logic than the one above but using generators. As it can be observed, the improvement on code readability is noticeable.

```
co(function*(){
 try{
   var player = yield pipeline.create('PlayerEndpoint');
   yield player.play()
   console.log("Play started");
 } catch(e) {
   onError(e);
 }
})();
```

The next version of JavaScript, ES7, which is still under standardization, has a proposal to simplify this: the async/await keyword, which marks when a call is asynchronous but accepts synchronous API syntax. Using it, the code in previous snippet can be written as shown below with ES7. As it can be observed, the yield keyword is replaced by await and the co function is no longer necessary.

```
try{
    var player = await pipeline.create('PlayerEndpoint');
    await player.play()
    console.log("Play started");
 } catch(e) {
   onError(e);
 }
```

5.2 NUBOMEDIA Repository API implementation

The interface `RepositoryClient` specifies the API to communicate with the repository server. It uses REST as means of communicating with the server. The factory to create instances of `RepositoryClient` is another class called `RepositoryClientProvider`. This factory requires the repository's service URL for

REST communications, which can be configured by means of reading properties from well-known locations (class `RepositoryUrlLoader`), or directly when instantiating the provider. This class hierarchy is illustrated on Figure 15.

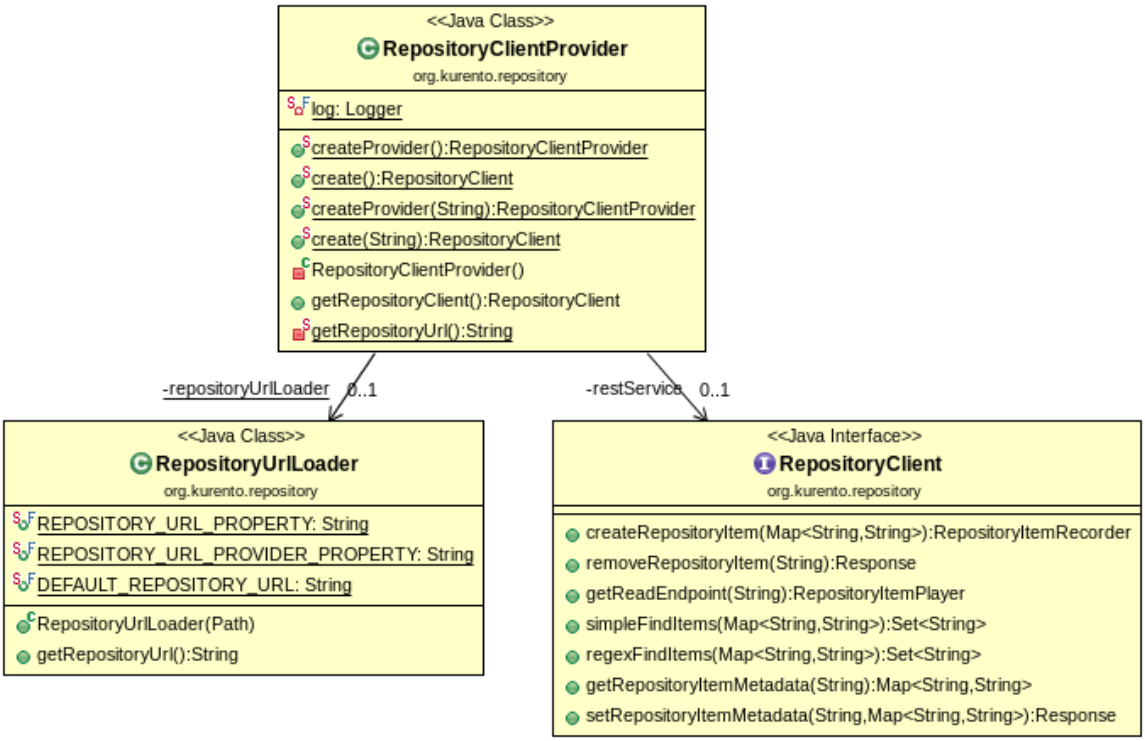


Figure 15. `RepositoryClient`class diagram

As introduced in section 4.3, a key concept in the repository API in the Repository Item. These items can be managed using CRUD operations, and are implemented as POJOs accessed by REST on the server-side. The root class for these items is declared on the Java class `RepositoryItemEndpoint`. This class is inherited by (see Figure 16):

- `RepositoryItemPlayer`, which represents an HTTP endpoint of a Repository Item that can be used for playing (retrieving) multimedia.
- `RepositoryItemRecorder`, which represents an HTTP endpoint of a Repository Item that can be used for recording multimedia.

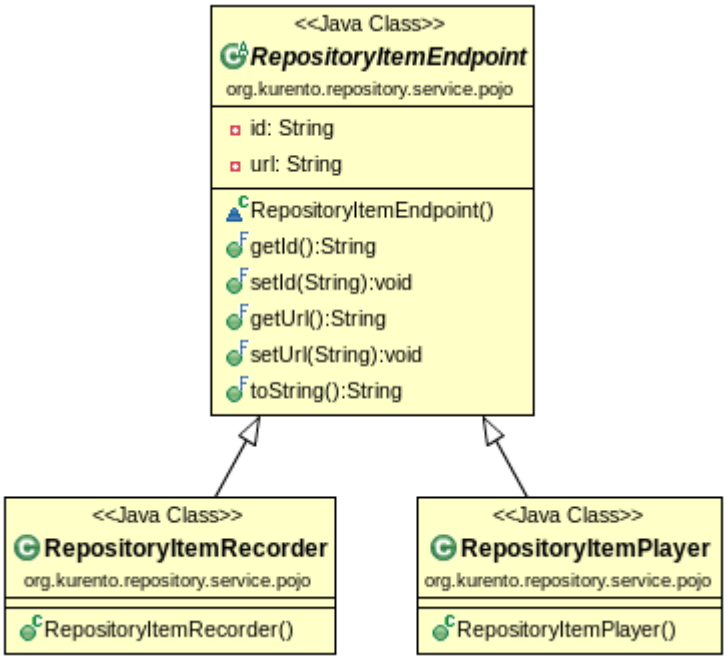


Figure 16. `RepositoryItem` class diagram

5.3 NUBOMEDIA WebRtcPeer API implementation

The NUBOMEDIA WebRtcPeer API implementation has three implementations one for each of the target clients platforms. The following sections are devoted to introducing them:

5.3.1 WebRTC browser WebRtcPeer API implementation

As depicted in section 4.4, the NUBOMEDIA WebRtcPeer API offers a `WebRtcPeer` object for the browser side. This object is a wrapper of the browser's RTCPeerConnection API, and its constructor is as follows:

```
WebRtcPeer(mode, options, callback)
```

…where:

- `mode`: Mode in which the PeerConnection will be configured. Valid values are
  - `recv`: receive only media.
  - `send`: send only media.
  - `sendRecv`: send and receive media.
- `options` : It is a group of parameters and they are optional. It is a json object.
  - `localVideo`: Video tag in the application for the local stream.
  - `remoteVideo`: Video tag in the application for the remote stream.
  - `videoStream`: Provides an already available video stream that will be used instead of using the media stream from the local webcam.
  - `audioStreams`: Provides an already available audio stream that will be used instead of using the media stream from the local microphone.
  - `mediaConstraints`: Defined the quality for the video and audio
  - `connectionConstraints`: Defined the connection constraint according with browser like googIPv6, DtlsSrtpKeyAgreement, ...
  - `peerConnection`: Use a peerConnection which was created before

- `sendSource`: Which source will be used: It can be one of the following `values`: webcam, screen, window
- `onstreamended`: Method that will be invoked when stream ended event happens
- `onicecandidate`: Method that will be invoked when ice candidate event happens
- `oncandidategatheringdone`: Method that will be invoked when all candidates have been harvested
- `simulcast`: Indicates whether simulcast is going to be used. Value is `true|false`
- `configuration`: It is a json object where ICE Servers are defined using
  - `iceServers`: The format for this variable is as follows:

```
[{
    "urls": "turn:turn.example.org",
    "username": "user",
    "credential": "myPassword"
}]
[{
    "urls": "stun:stun1.example.net"
}, {
    "urls": "stun:stun2.example.net"
}]
```

- callback: It is a callback function which indicate, if all worked right or not

Also there are 3 specific methods for creating `WebRtcPeer` objects without using mode parameter:

- `WebRtcPeerRecvonly(options, callback)`: Create a `WebRtcPeer` as receive only.
- `WebRtcPeerSendonly(options, callback)`: Create a `WebRtcPeer` as send only.
- `WebRtcPeerSendrecv(options, callback)`: Create a `WebRtcPeer` as send and receive.

Constraints provide a general control surface that allows applications to both select an appropriate source for a track and, once selected, to influence how a source operates. `getUserMedia()` uses constraints to help select an appropriate source for a track and configure it. For more information about media constraints and its values, you can check here. By default, if the `mediaConstraints` is undefined, these constraints are used when `getUserMedia` is called:

```
{
  audio: true,
  video: {
    width: 640,
    framerate: 15
  }
}
```

The methods of `WebRtcPeer` are the following:

- `getPeerConnection`: Using this method the user can get the peerConnection and use it directly.
- `showLocalVideo`: Use this method for showing the local video.
- `getLocalStream`: Using this method the user can get the local stream. You can use muted property to silence the audio, if this property is true.
- `getRemoteStream`: Using this method the user can get the remote stream.
- `getCurrentFrame`: Using this method the user can get the current frame and get a canvas with an image of the current frame.
- `processAnswer`: Callback function invoked when a SDP answer is received. Developers are expected to invoke this function in order to complete the SDP negotiation. This method has two parameters:
  - `sdpAnswer`: Description of the SDP answer
  - `callback`: It is a function with error like parameter. It is called when the remote description has been set successfully.
- `processOffer`: Callback function invoked when a SDP offer is received. Developers are expected to invoke this function in order to complete the SDP negotiation. This method has two parameters:
  - `sdpOffer`: Description of the SDP offer
  - `callback`: It is a function with error and `sdpAnswer` like parameters. It is called when the remote description has been set successfully.
- `dispose`: This method frees the resources used by `WebRtcPeer`.
- `addIceCandidate` Callback function invoked when an ICE candidate is received. Developers are expected to invoke this function in order to complete the SDP negotiation. This method has two parameters:
  - `iceCandidate`: Literal object with the ICE candidate description
  - `callback`: It is a function with error like parameter. It is called when the ICE candidate has been added.
- `getLocalSessionDescriptor`: Using this method the user can get peerconnection's local session descriptor.
- `getRemoteSessionDescriptor`: Using this method the user can get peerconnection's remote session descriptor.
- `generateOffer`: Creates an offer that is a request to find a remote peer with a specific configuration.

5.3.2 Android WebRtcPeer API implementation

This chapter introduces how to use webRTC communication in Android applications. Generally speaking, WebRTC applications need to do several things:

- Get streaming audio, video or other data.
- Get network information such as IP addresses and ports, and exchange this with other WebRTC clients (known as peers) to enable connection, even through NATs and firewalls.
- Coordinate signalling communication to report errors and initiate or close sessions.
- Exchange information about media and client capability, such as resolution and codecs.
- Communicate streaming audio, video or data.

The Androidn webRTC API consists of following interface libraries:

- AppRTCClient (interface)
- WebSocketRTCClient (Class which implements AppRTCClient interface)
- PeerConnectionClient (Class implementing the peer connection client)
- LooperExecutor (An executor class which implements Android Executor Class and utilizes Looper classes)

APPRTCClient.java is an interface representing an AppRTC client. It has the following methods:

Return value	Method	Description
void	connectToServer(String url)	Asynchronously connect to an AppRTC room URL using supplied connection parameters. Once connection is established onConnectedToRoom() callback with room parameters is invoked.
void	sendOfferSdp(final SessionDescription sdp)	Send offer SDP to the other participant.
void	sendAnswerSdp(final SessionDescription sdp)	Send answer SDP to the other participant.
void	sendLocalIceCandidate(final IceCandidate candidate)	Send Ice candidate to the other participant.
void	disconnectFromServer();	Disconnect from room.

There is also a callback interface (SignalingEvents), which is used for messages delivered on signaling channel. Interface has a list of methods which can be seen below.

Return value	Method	Description
void	onConnectedToServer(final SignalingParameters params	Callback fired once the room's signaling parameters SignalingParameters are extracted.
void	onRemoteDescription(final SessionDescription sdp)	Callback fired once remote SDP is received.
void	onRemoteIceCandidate(final IceCandidate candidate)	Callback fired once remote Ice candidate is received.
void	onChannelClose()	Callback fired once channel is closed.
void	onChannelError(final String description)	Callback fired once channel error happened.

In addition, it also has a struct called SignalingParameters to hold signaling parameters, such as the list of iceServers, offerSdp and list of iceCandidates. This struct only has a constructor, which is used to set parameters:

Return value	Method	Description
constructor	SignalingParameters(List<PeerConnection.IceServer> iceServers, SessionDescription offerSdp, List<IceCandidate> iceCandidates)	Struct holding the signaling parameters of an AppRTC room.

WebSocketRTCClient.java implements the AppRTCClient interface.

PeerConnectionClient.java is an implementation of peer connection client. For developers, it provides means for easy video formatting, setting signaling parameters and configuring media constraints. With this class, developer can easily set all the necessary parameters and create a webRTC peer connection. This class includes the following methods:

Return value	Method	Description
constructor	PeerConnectionParameters(boolean videoCallEnabled, int videoWidth, int videoHeight, int videoFps, int videoStartBitrate, String videoCodec, boolean videoCodecHwAcceleration, boolean cpuOveruseDetection)	Constructor to set initial values
void	setPeerConnectionFactoryOptions(PeerConnectionFactory.Options options)	Used to change PeerConnectionFactory.Options
void	createPeerConnectionFactory(final Context context, final PeerConnectionParameters peerConnectionParameters, final PeerConnectionEvents events)	Creates PeerConnectionFactory with the PeerConnectionParameters and PeerConnectionEvents
void	createPeerConnection(final EGLContext renderEGLContext, final VideoRenderer.Callbacks localRender, final VideoRenderer.Callbacks remoteRender, final SignalingParameters signalingParameters)	This method is used to create a peer connection. It is necessary to set videorenderer callbacks and signalingparameters as a method parameter.
void	close()	This method is used to close the video connection. The method closes peer connection, video source and peer connection factory. Finally it calls the onPeerConnectionClosed event.
boolean	isVideoCallEnabled()	Used to check whether the video call is enabled or not.
boolean	isHDVideo()	Used to check whether the HD video is envabled or not.
void	setVideoEnabled(final boolean enable)	Used to enable/disable the video
void	createOffer()	This method creates SDPOffer
void	createAnswer()	This method creates SDPAnswer
void	addRemoteIceCandidate(final IceCandidate candidate)	Used to set iceCandidate
void	setRemoteDescription(final SessionDescription sdp)	Used to set SessionDescription
void	stopVideoSource()	Used to stop video source
void	startVideoSource()	Used to start video source
void	switchCamera()	Used to switch the device's camera used in video call.

in Addition, class has an interface (PeerConnectionEvents) for the following peer connection events:

Return value	Method	Description
void	onLocalDescription(final SessionDescription sdp)	Callback fired once local SDP is created and set.
void	onIceCandidate(final IceCandidate candidate)	Callback fired once local Ice candidate is generated.
void	onIceConnected()	Callback fired once connection is established (IceConnectionState is CONNECTED)
void	onIceDisconnected()	Callback fired once connection is closed (IceConnectionState is DISCONNECTED
void	onPeerConnectionClosed()	Callback fired once peer connection is closed.
void	onPeerConnectionError(final String description)	Callback fired once peer connection error happened.

LooperExecutor.java is a just a class which implements Android Excecutor class. It helps executing the asynchronous websocket connections. WebSocketRTCClient.java utilizes the LooperExecutor for its web connections.

5.3.3 iOS WebRtcPeer API implementation

Software architecture

The iOS API implementation, named Kurento Toolbox for iOS, basically helps iOS application developers in setting up and maintaining the link with Kurento without knowing in depth the mechanisms standing behind Kurento and JSON-RPC communication. It provides a set of basic components that have been found useful during the native development of the WebRTC applications with Kurento. The idea is to build a communication layer to be included into the app, offering methods to higher layers.

A demo application was developed in order to show the usage of some of the functionalities provided by the toolkit. It includes two modules of this kind:

- JSON-RPC 2.0 client over WebSocket for signaling (see 5.4.2 for further details).
- WebRTCPeer manager aimed to simplify WebRTC interactions (i.e. handle multiple peer connections, manage SDP offer-answer dance, retrieve local or remote streams etc). This class is implemented based on the native WebRTC library libjingle peerconnection.

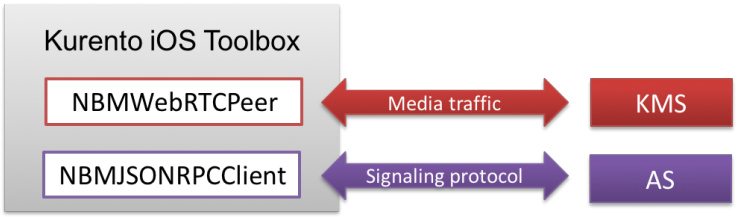


Figure 17. iOS WebRtcPeer API modules

Classes related to the WebRTC module are illustrated on Figure 18. This module is responsible for handling WebRTC streams on the client side. In this case the main component that makes this possible is represented by the `NBMWebRTCPeer`, this manager provides several methods to facilitate SDP and ICE negotiation between client and KMS, supporting multiple remote streams. This last feature is essential in contexts when it is required to enable users to simultaneously establish multiple connections to other users connected to the same session or room. In order to setup a peer connection in a more robust way the module includes an improved version of `RTCPeerConnection`, it wraps this base class of the libjingle library providing better support at different stages of its use and configuration.

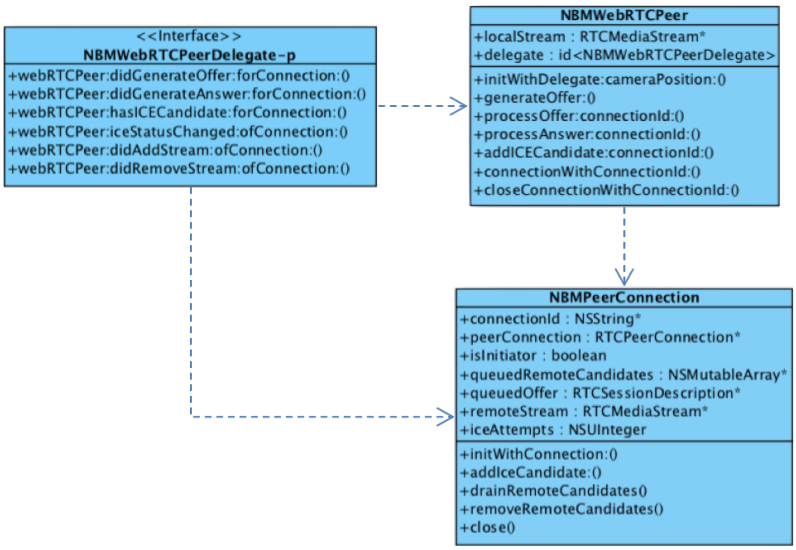


Figure 18. iOS WebRtcPeer API JSON-RPC WebRTC class diagram

Implementation details

As anticipated in the "Software architecture" paragraph above, the development of the toolkit draws inspiration from existing projects around Kurento's ecosystem.
Starting from JSON-RPC module, it was designed following the structure of the project containing the Javascript client library for JSON-RPC, whose implementation can be found on GitHub at this link: https://github.com/Kurento/kurento-jsonrpc-js. The toolkit basically follows the logic found in that project, especially in proper control of timeouts and attempts to resend request in case of error.
The module makes use of two third party libraries:

- SBJson 4, an open-source framework that implements a strict JSON parser and generator in Objective-C, available at: https://github.com/stig/json-framework
- SocketRocket, the best open-source WebSocket client library for iOS, it conforms to almost all of the base Autobahn test suite, available at: https://github.com/square/SocketRocket.

Beside this, for the WebRTC part, the toolkit depends in large part on the library called libjingle_peerconnection, which is specified here: http://www.webrtc.org/native-code/ios.

Kurento Toolbox requires iOS version 8 or later and Armv7/Arm64 architectures and it does not offer full support for iOS 32/64 bit simulators (since there is no access to camera yet).

In terms of Apple libraries, next is reported the dependencies list:

- AVFoundation.framework
- AudioToolbox.framework
- CoreGraphics.framework
- CoreMedia.framework
- GLKit.framework
- UIKit.framework

- VideoToolbox.framework
- CFNetwork.framework
- Security.framework

Evaluation and validation

The implementation provided is ready to be linked to a continuous integration system like Jenkins or Travis CI. The continuous integration system will check the consistency by performing the tests especially on JSON-RCP module:

- NBMJSONRPCClientTests
- NBMTransportChannelTests

Information for developers

The Kurento Toolkit for iOS, will soon be available for integration via CocoaPods dependency manager. Until then it is possible to manually install it by following this procedure:

- Step 1. Download and unzip the framework from https://github.com/nubomediaTI/Kurento-iOS/releases/download/v0.2/KurentoToolbox.framework.zip
- Step 2. Drag The Kurento Toolbox Framework onto the target Xcode project, meanwhile make sure the "Copy items to destination's group folder" checkbox is checked.
- Step 3. Link Binary With Library Framework, taking into account the apple library dependencies specified above.
- Step 4. Add –Objc linker flag
- Step 5. Import headers file by sampling using the directive

```
#import <KurentoToolbox/KurentoToolbox.h>
```

A more detailed documentation explaining the toolkit usage and integration process can be found at http://kurento-ios.readthedocs.org/en/latest/, the source code for the toolkit is provided on GitHub as part of the Nubomedia project at this link: https://github.com/nubomediaTI/Kurento-iOS

With respect to the source code, some Apple Style Documentation of API is provided at http://rawgit.com/nubomediaTI/Kurento-iOS/master/docs/html/index.html

5.4 NUBOMEDIA Signaling API implementation

The implementation of the NUBOMEDIA Signaling API can be split into two parts: server-side and client-side. The following sections show each part.

5.4.1 NUBOMEDIA Signaling API implementation server-side

This section summarizes the Java implementation of the signaling API in the server-side. The server-side component listening to signaling messages has been implemented using Spring Boot. The usage is very simple, and analogous to the creation and

configuration of a WebSocketHandler from Spring. It is basically composed of the server's configuration, and a class that implements the handler for the requests received. This server has been published as a Maven artifact, allowing developers to easily manage it as a dependency, by including the following dependency in their project's pom:

```
<dependency>
    <groupId>org.kurento</groupId>
    <artifactId>kurento-jsonrpc-server</artifactId>
</dependency>
```

As depicted in section 4.5, JSON-RPC v2.0 messages are exchange between client and server. The components in charge of managing the request messages from the client are called handler, which are implemented as Java classes extending `DefaultJsonRpcHandler<JsonObject>`. This is generified with the payload that comes with the request. In this case is a `JsonObject`, but it could also be a plain String, or any other object. The methods available for handlers are the following:

- `sendResponse`: sends a response back to the client.
- `sendError`: sends an Error back to the client.
- `getSession`: returns the JSON-RPC session assigned to the client.
- `startAsync`: in case the programmer wants to answer the Request outside of the call to the handleRequest method, he can make use of this method to signal the server to not answer just yet. This can be used when the request requires a long time to process, and the server not be locked.
- `isNotification`: evaluates whether the message received is a notification, in which case it mustn't be answered.

When a class inherits from the parent class `DefaultJsonRpcHandler`, it is mandatory to override the method `handleRequest` (see section 4.5.3). This method can be used to access any of the fields from a JSON-RPC request. This is where the methods invoked should be managed. Besides the methods processed in this class, the server handles also the following special method values:

- `close`: The client send this method when gracefully closing the connection. This allows the server to close connections and release resources.
- `reconnect`: A client that has been disconnected, can issue this message to be attached to an existing session. The `sessionId` is a mandatory parameter.
- `ping`: simple ping-pong message exchange to provide heartbeat mechanism.

5.4.2 NUBOMEDIA Signaling API implementation client-side

This part can be also divided in different implementations, namely:

- WWW implementation
- Android implementation
- iOS implementation

Regarding the **WWW signaling implementation**, contrary to the server, the client is framework-agnostic, so it can be used in regular Java applications, Java EE, Spring, and so on. Creating a client that will send requests to a certain server is very straightforward. The URI of the server is passed to the `JsonRpcClientWebSocket` in the constructor (see section 4.5.3).

The client offers the possibility to set-up a listener for certain connection events. A user can define a `JsonRpcWSConnectionListener` that offers overrides of certain methods. Once the connection listener is defined, it can be passed in the constructor of the client, and the client will invoke the methods once the corresponding events are produced:

```
JsonRpcWSConnectionListener listener = new JsonRpcWSConnectionListener() {

    @Override
    public void reconnected(boolean sameServer) {
        // ...
    }

    @Override
    public void disconnected() {
        // ...
    }

    @Override
    public void connectionFailed() {
        // ...
    }

    @Override
    public void connected() {
        // ...
    }
} ;

JsonRpcClient client = new
      JsonRpcClientWebSocket("ws://localhost:8080/echo", listener);
```

Android signaling

Jsonrpc-ws-android is an Android-library for sending JSON-RPC style messages over a Web Socket transfer protocol. Its source code is available at https://github.com/nubomedia-vtt/jsonrpc-ws-android.
The Maven artefact is available at http://mvnrepository.com/artifact/fi.vtt.nubomedia/jsonrpc-ws-android. The library is a set of wrapper classes utilizing JSON-RPC 2.0 Client available at http://software.dzhuvinov.com/json-rpc-2.0-client.html. It consists of the classes described in Table 3.

Class	Description
JsonRpcWebSocketClient	Handles the Web Socket connection, sending requests, and receiving requests, notifications and errors.
JsonRpcRequest	Is a placeholder for request's id, method, named parameters and positional parameters.
JsonRpcResponse	Is a placeholder for responses id, result, error and indicator if the result was successful.
JsonRpcNotification	Is a placeholder for notification's method, named parameters and positional parameters.
JsonRpcResponseError	Is a placeholder for error message's error code and data.

Regarding the iOS signaling implementation, the application does not dialog directly with the Media Server, but provides a JSON-RPC 2.0 over WebSocket signaling channel linked to the AS that communicates with the Media Server.

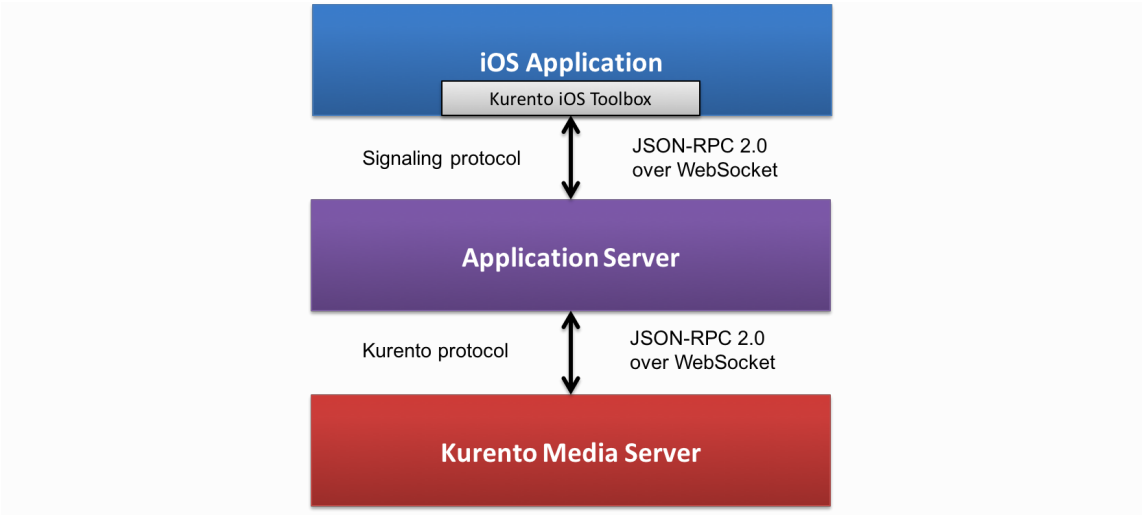


Figure 19. iOS WebRtcPeer API schema

The JSON-RPC module of the iOS API is responsible for the signaling. This module provides factory methods that create Message objects fully compatible with the JSON-RPC 2.0 specification. The base protocol `NBMMessage` guarantees that these message types can be serialized to JSON string, which is the format used by the transport channel. The Message objects can be obtained by passing positional (JSON array) or named (JSON object) parameters, or directly with the `NSDictionary` representation of JSON message. There is no need to include an identifier when creating a request; this is inflated automatically by the JSON-RPC client (`NBMJSONRPCClient`) during the message scheduling

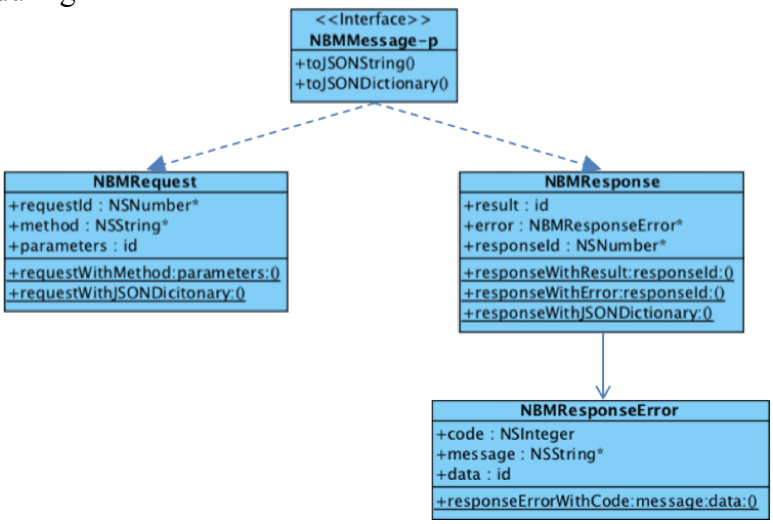


Figure 20. iOS WebRtcPeer API JSON-RPC `NBMMessage` class diagram

This other part of the JSON-RPC module is related to the `NBMJSONRCPClient`, a client-side class for dispatching requests and notifications to a JSON-RPC 2.0 server using WebSocket channel (`NBMTransportChannel`). The client class has a number of configurable properties that allow changing the timeout of requests and resend them in case of failure. A Response object is received as an argument of an asynchronous block

after calling `sendRequest`-like API, no block is executed when sending notification as expected.

The client defines a formal `NBMJSONRPCClientDelegate` protocol to notify its delegate about occurrence of relevant events, such as the clean initialization of the communication layer or the arrival of a Notification from the server; the implementation of these methods is required in order to properly use this class.

The messages are dispatched by means of WebSocket `send()` and this is done under the hood by `NBMTransportChannel`. The client takes charge to keep connection alive, monitoring the channel status and restore it if necessary.

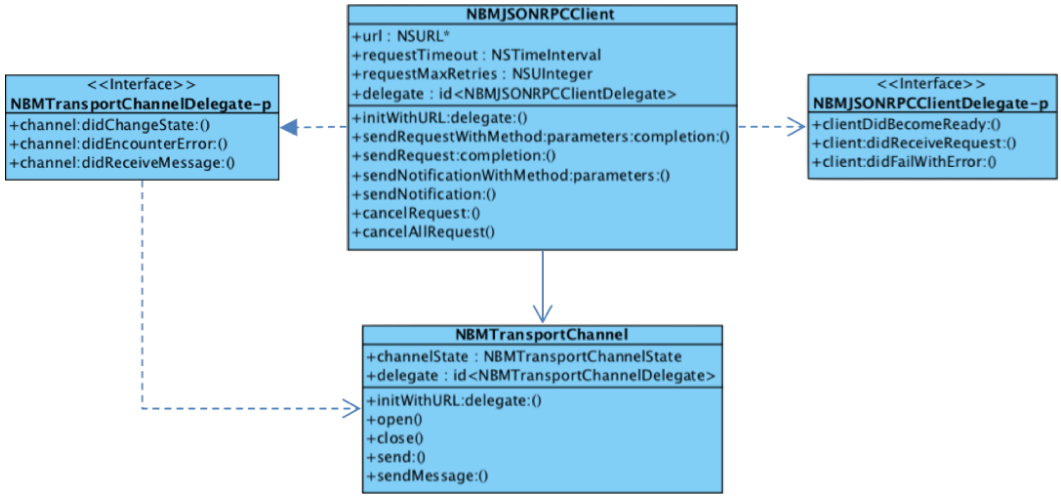


Figure 21. iOS WebRtcPeer API JSON-RPC `NBMJSONRPCClient` class diagram

5.5 NUBOMEDIA Room API implementation

As introduced in section 4.6, the NUBOMEDIA Room API should be understood as an SDK for any developer that wants to implement a Room client-server application. Regarding the **server-side**, the API is based on the Room Manager abstraction. This manager can organize and control multi-party group calls.

The Room Manager's Java API takes care of the room and media-specific details, freeing the programmer from low-level or repetitive tasks (inherent to every multi-conference application) and allowing her to focus more on the application's functionality or business logic. A new room manager should provide implementation for:

- The Room Handler in charge of events triggered by internal media objects
- A Client Manager that will be used to obtain instances of Kurento Client (client to access to the NUBOMEDIA Media API)

Regarding the Room Handler, in order to act upon events raised by media objects, such as new ICE candidates gathered or media errors, the application has to provide an event handler. Generally speaking, these are user-orientated events, so the application should notify the corresponding users. The following is a table detailing the server events that will resort to methods from Room Handler.

Events	Room Handler

Gathered ICE candidate	`onSendIceCandidate`
Pipeline error	`onPipelineError`
Media element error	`onMediaElementError`

The notification managing API considers two different types of methods:

- Server domain - consists of methods designed to be used in the implementation of the application's logic tier and the integration with the room SDK. The execution of these methods will be performed synchronously. They can be seen as helper or administration methods and expose a direct control over the rooms.
- Client domain - methods invoked as a result of incoming user requests, they implement the room specification for the client endpoints. They could execute asynchronously and the caller should not expect a result, but use the response handler if it's required to further analyze and process the client's request.

The following diagram describes the components that make up the system when using the notifications room manager:

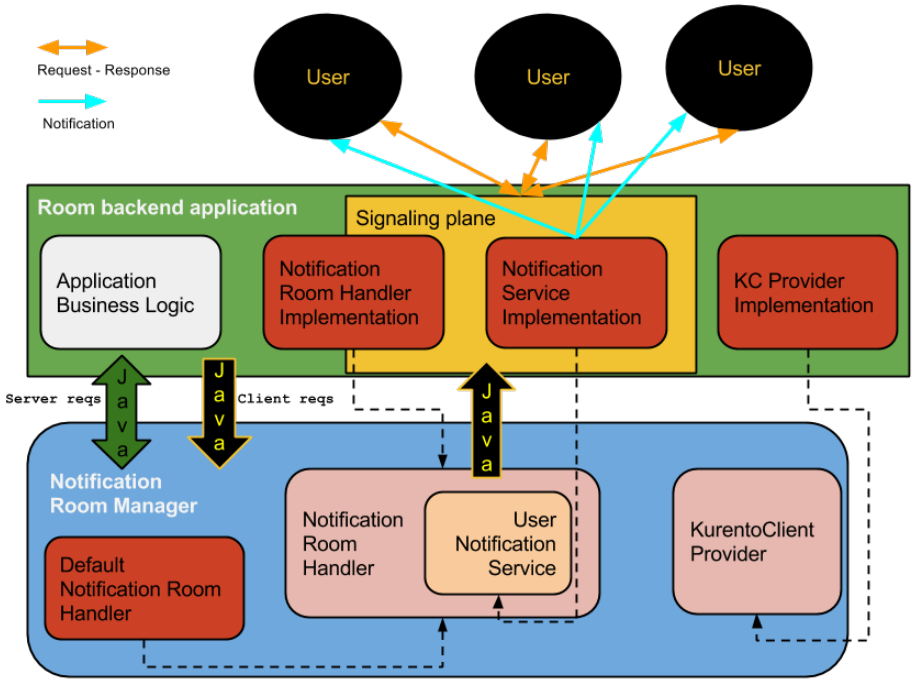


Figure 22. iOS Notification Room Manager

Regarding the **client-side**, in order to a Room Server it is required to create an instance of `KurentoRoomClient` class indicating the URI of the application server's WebSocket endpoint:

```
KurentoRoomClient client = new
      KurentoRoomClient("wss://roomAddress:roomPort/room");
```

In background, a websocket connection is made between the Java application and the Kurento Room Server.

As the client is no more than a wrapper for sending and receiving the messages defined by the Room Server's WebSocket API, the methods of this API are quite easy to understand (as they reflect the JSON-RPC messages).

The client maintains a notifications' queue where it stores messages received from the server. The developer should run the following method in a separate thread using an infinite loop:

```
Notification notif = client.getServerNotification();
```

The `Notification` abstract class publishes a method that can be used to find its exact type:

```
if (notif == null)
   return;
log.debug("Polled notif {}", notif);
switch (notif.getMethod()) {
   case ICECANDIDATE_METHOD:
      IceCandidateInfo info = (IceCandidateInfo) notif;
      //do something with the ICE Candidate information
      ...
      break;
   ...
}
```

The notification types are the following and they contain information for the different types of events triggered from the server-side:

- `org.kurento.room.client.internal.IceCandidateInfo`
- `org.kurento.room.client.internal.MediaErrorInfo`
- `org.kurento.room.client.internal.ParticipantEvictedInfo`
- `org.kurento.room.client.internal.ParticipantJoinedInfo`
- `org.kurento.room.client.internal.ParticipantLeftInfo`
- `org.kurento.room.client.internal.ParticipantPublishedInfo`
- `org.kurento.room.client.internal.ParticipantUnpublishedInfo`
- `org.kurento.room.client.internal.RoomClosedInfo`
- `org.kurento.room.client.internal.SendMessageInfo`

The following table summarizes the rest of operations that can be executed from the `client` instance:

Operation	Description	Example
Join room	This method sends the joinRoom message and returns a list containing the existing participants and their published streams.	`Map<String, List<String>> newPeers = client.joinRoom(room, username);`
Leave room	This method sends the leaveRoom message.	`client.leaveRoom();`
Publish	This method sends the publishVideo	`String sdpAnswer =`

	message. It returns the SDP answer from the publishing media endpoint on the server.	`client.publishVideo(sdpOffer, false);`
Unpublish	This method sends the unpublishVideo message.	`client.unpublishVideo();`
Subscribe	This method sends the receiveVideoFrom message. It returns the SDP answer from the subscribing media endpoint on the server.	`String sdpAnswer = client.receiveVideoFrom(sender, sdpOffer);`
Unsubscribe	This method sends the unsubscribeFromVideo message.	`client.unsubscribeFromVideo(sender);`
Send ICE Candidate	This method sends the onIceCandidate message, containing a local ICE Candidate for the connection with the specified endpoint.	`client.onIceCandidate(endpointName, candidate, sdpMid, sdpMLineIndex);`
Send message	This method sends the sendMessage message.	`client.sendMessage(userName, roomName, message);`

5.6 NUBOMEDIA Tree API implementation

Currently, there are no binary releases of Tree Server. In order to deploy a new Tree Server it is needed to build it from sources. The software requirements in that case are the following:

- JDK 7 or 8
- Git
- Maven
- Bower, NPM, and Node.js
- Kurento Media Server or connection with at least a running instance

When all dependencies are installed, the following Git repositories have to be cloned. Then the server should be executed:

```
git clone https://github.com/Kurento/kurento-java.git
cd kurento-java
mvn install -DskipTests=true
cd ..
git clone https://github.com/Kurento/kurento-tree.git
cd kurento-tree
mvn install -DskipTests=true
cd kurento-tree-server
mvn exec:java
```

Then a bunch of log messages will appear in the console. When the following message appears in the console, there should be an instance of the Tree Server up and running (by default this server is listening to client requests in the port 8890):

```
Started KurentoTreeServerApp in 4.058 seconds (JVM running for 8.017)
```

Regarding the **JavaScript Client**, the library files needed to use this client are served by Tree Server in the URL: http://server-host:port/js/KurentoTree.js

In addition, there are several third-party libraries that need to be imported in the HTML in order to use this JavaScript client. These libraries are also served by Kurento Tree Server in the paths:

- http://treeserver:port/bower_components/adapter.js/adapter.js
- http://treeserver:port/bower_components/eventEmitter/EventEmitter.js
- http://treeserver:port/js/kurento-utils.js
- http://treeserver:port/lib/sockjs.js
- http://treeserver:port/js/websocketwithreconnection.js
- http://treeserver:port/js/kurento-jsonrpc.js
- http://treeserver:port/js/jsonRpcClient.js

For example, all necessary dependencies to create a SPA Tree application can be included in the HTML with the elements:

```
<script
src="http://treeserver:port/bower_components/adapter.js/adapter.js"></script>
<script
src="http://treeserver:port/bower_components/eventEmitter/EventEmitter.js"></
script>
<script src="http://treeserver:port/js/kurento-utils.js"></script>
<script src="http://treeserver:port/lib/sockjs.js"></script>
<script
src="http://treeserver:port/js/websocketwithreconnection.js"></script>
<script src="http://treeserver:port/js/kurento-jsonrpc.js"></script>
<script src="http://treeserver:port/js/jsonRpcClient.js"></script>
<script src="http://treeserver:port/js/KurentoTree.js"></script>
```

When these files are included in the HTML, the JavaScript class `KurentoTree` becomes available. This class is the entry point of the Tree JavaScript Client. To connect to a Tree Server it is necessary to create an instance of `KurentoTree` class indicating the URL of the server:

```
var tree = new KurentoTree('http://treeserver:port/kurento-tree');
```

In background, a WebSocket connection is made between browser and Tree server. To broadcast the user webcam it is necessary to execute a code similar to the following:

```
var treeName = ...

var mediaOptions = {
   localVideo : video,
   mediaConstraints : {
      audio : true,
      video : {
         mandatory : {
            maxWidth : 640,
            maxFrameRate : 15,
            minFrameRate : 15
         }
      }
   }
};

tree.setTreeSource(treeName, mediaOptions);
```

Where `localVideo` refers to a HTML video element in which the local video is shown. If this option is not specified, no local video will be shown in the page. In order

to include a player showing the video that it is broadcasting another user, it is necessary to include a code similar to:

```
var treeName = ...

var mediaOptions = {
   remoteVideo : video
}

tree.addTreeSink(treeName, options);
```

Where `remoteVideo` refers to the HTML video element in which remote video is shown. To stop any transmission (from emitter or receiver), the `close()` method can be invoked:

```
tree.close();
```

Regarding the **Java Client**, this library gives more control to developer and allows including authentication and authorization to broadcast applications. To create broadcast applications with this Java client, it is necessary to implement frond-end logic in JavaScript that communicates with Java web applications using WebSocket or another technology. Then, Java back-end will communicate with Tree Server using this client. This client can be obtained as a maven dependency with the following Maven coordinates:

```
<dependency>
   <groupId>org.kurento</groupId>
   <artifactId>kurento-tree-client</artifactId>
</dependency>
```

With this dependency, the developer can use the class `org.kurento.tree.client.KurentoTreeClient` to control the Tree Server. To connect to a Tree Server it is necessary to create an instance of `KurentoTreeClient` class indicating the URL of the server:

```
KurentoTreeClient tree =
    new KurentoTreeClient("http://treeserver:port/kurento-tree");
```

In background, a WebSocket connection is made between Java app and Kurento Tree server. To broadcast the user webcam it is necessary to execute a code similar to the following:

```
String treeName = ... // Select a unique name for the broadcast
tree.createTree(treeName);
String sdpOffer = ... // Create sdp offer in browser
String sdpAnswer = tree.setTreeSource(treeName, sdpOffer);
// Send sdpAnswer to browser to complete media negotiation
```

In this code, `sdpOffer` have to be created in the browser and communicated to Java web app using WebSocket or other technology. On the other hand, `sdpAnswer` have to be sent to browser.

```
String treeName = ... // The broadcast name
String sdpOffer = ... // Create sdp offer in browser
```

```
TreeEndpoint treeEndpoint = tree.addTreeSink(treeName, sdpOffer);
String viewerId = treeEndpoint.getId(); // Id of this viewer
String sdpAnswer = treeEndpoint.getSdp();
// Send sdpAnswer to browser to complete media negotiation
```

In same way as before, `sdpOffer` and `sdpAnswer` have to be interchanged with JavaScript code in the browser to perform the media negotiation. The negotiation is done with Trickle ICE[2] and so, besides SDP offer and answer interchange between browser and media server, it is necessary to interchange ICE candidates between peers.

```
while (true) {
   // Retrieve new ice candidate from server
   IceCandidateInfo cand = tree.getServerCandidate();

   if (candidateInfo == null) {
      // No more ice candidates. Connection closed
      break;
   }

   // Tree to which belongs this candidate
   String treeName = cand.getTreeId();

   // Viewer to which belongs this candidate (if null, it is tree source)
   String viewerId = cand.getSinkId();

   // Ice candidate info
   String candidate = cand.getIceCandidate().getCandidate());
   int sdpMLineIndex = cand.getIceCandidate().getSdpMLineIndex());
   String sdpMid = cand.getIceCandidate().getSdpMid());

   // Send candidate info to browser to complete media negotiation
}
```

When a new ice candidate is received from browser it is necessary to process it properly to achieve a successful media negotiation. This is done using the following code:

```
String treeName = ...
String viewerId = ... // null if is tree source

String candidate = ...
int sdpMLineIndex = ...
String sdpMid = ...

tree.addIceCandidate(treeName, viewerId,
    new IceCandidate(candidate, sdpMid, sdpMLineIndex));
```

6 NUBOMEDIA framework tools

6.1 NUBOMEDIA Visual Development Tool

6.1.1 Software architecture

The NUBOMEDIA Visual Development Tool is composed of two different applications:

- The Graph Editor is a desktop app that lets a developer visually create and edit component graphs and save them as json-based files with the `.ngeprj` file extension.
- The Code Generator is a command-line app that takes `.ngeprj` files and generates source code to implement these graphs in new applications.

The overall architecture is similar to that of Flux, as illustrated in the following figure:

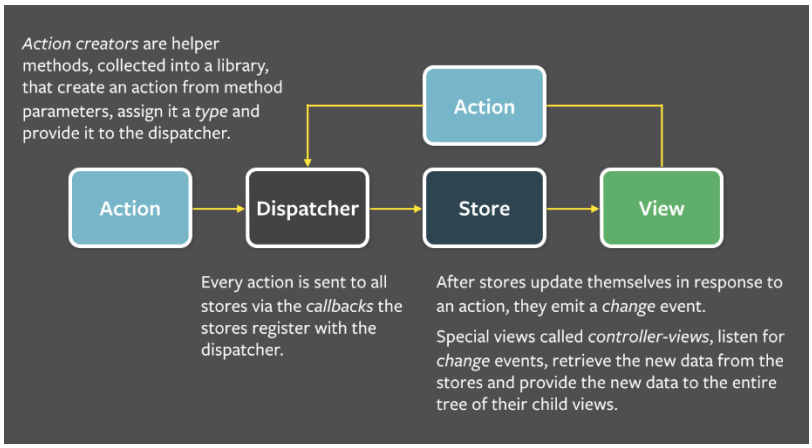


Figure 23. Flux architecture

Flux is an "Application Architecture for Building User Interfaces", built by the team at Facebook. It is a set of patterns building larger applications on top of the React component library.

6.1.2 Implementation details

Graph Editor

The Graph Editor is an HTML app based on Facebook's React and Dan Abramov's Redux libraries. The main gist of these technologies is to control the flow of data and mutations in an application.

React allows the UI and visual aspects of the application be defined purely as a function of the application state. This way, the UI is refreshed and reflects the changes to the app state automatically, rather than mutations of state needing on-the-fly in-place mutation of the corresponding pieces of the UI.

Redux is a variation of the Flux architecture proposed by Facebook as a companion to React, and is heavily inspired by functional UI frameworks such as Elm. It stores the entire application state in a single object (called the `store`), and this state must always be mutated in special functions called `reducers`, in response to dispatched `actions`. Mutation is a wrong word in fact, as Redux reducers are expected to create a *new* store containing the new state as required by the action they are reducing. This ensures a very simple, predictable and traceable flow of data and state throughout the entire application.

With these architectural pieces, it is easy to describe the application in terms of the store structure, and the components making up the UI. However, getting to grips with this style of architecture takes time, and the development of this project will reflect that learning process. Evolution of the tool will mean that some notable changes and request refactors will appear in the coming future.

Store

The Store is largely a reflection of the contents of the loaded Graph Editor project. It contains the following sub-stores:

- **nodedefs**: contains the loaded Node Definitions that are available as nodes for the user to insert in graphs
- **graphs**: contains the different graphs in the project
- **editor**: this is distinctly not part of the data saved in a project, but rather reflects the current state of the interactive editor: graph being currently edited, etc

NUBOMEDIA Graph Editor

1. Starting screen

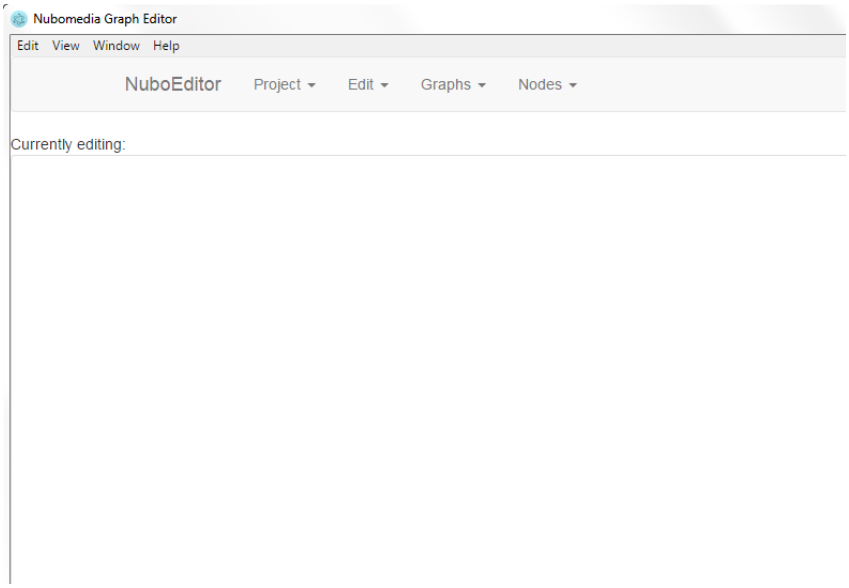


Figure 24. NUBOMEDIA Graph Editor. Step 1 - home screen

2. Load an **existing** Project

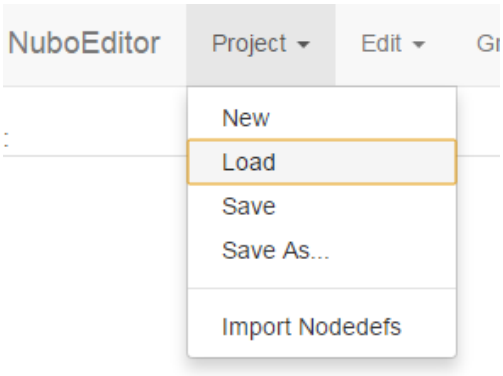


Figure 25. NUBOMEDIA Graph Editor. Step 2 - loading an existing project

3. Create a **New** Project

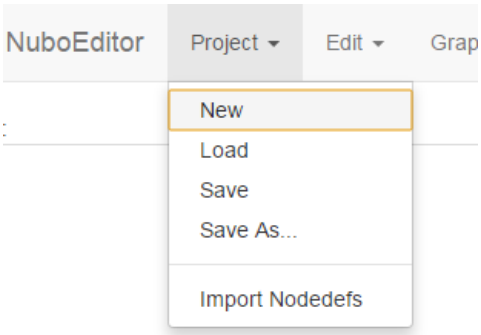


Figure 26. NUBOMEDIA Graph Editor. Step 3 - creating a new project

4. Working with a graph

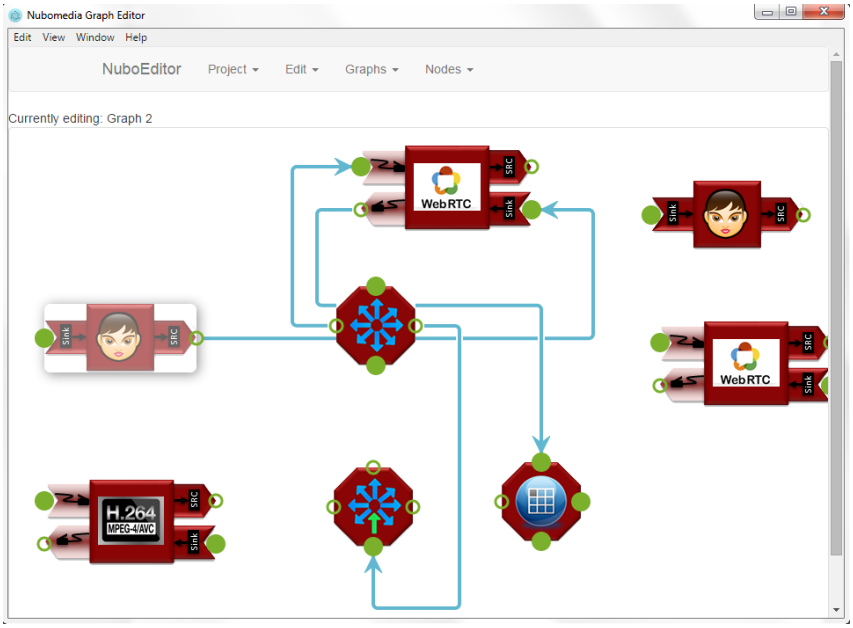


Figure 27. NUBOMEDIA Graph Editor. Step 4 - graph creation

5. **Saving** work on a graph

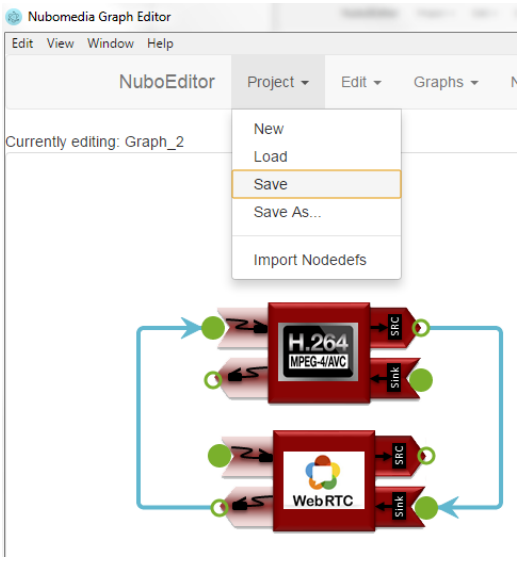


Figure 28. NUBOMEDIA Graph Editor. Step 5 – saving a graph

6. Possible actions while working on a graph: **Cut**, **Copy**, **Paste**, **Delete** can be performed on a given element (node or connection)

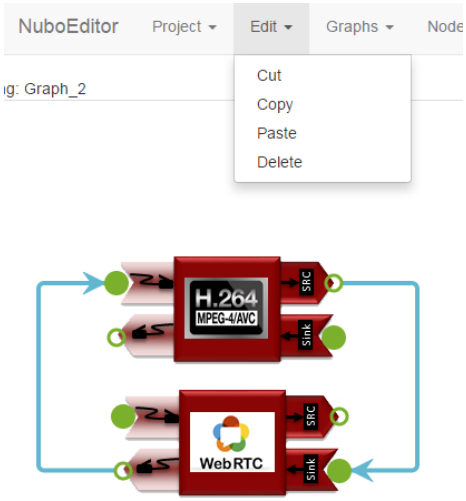


Figure 29. NUBOMEDIA Graph Editor. Step 6 - edit options

7. Nodes available

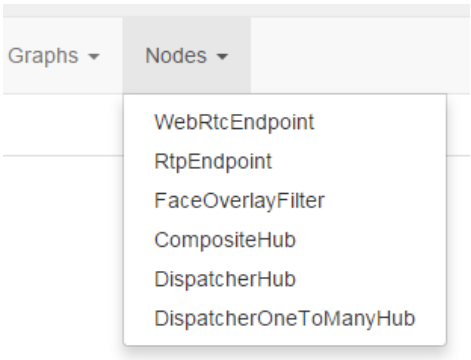


Figure 30. NUBOMEDIA Graph Editor. Step 7 – nodes availability

Caveats

This application uses *jsPlumb* as the library to manage display and editing of graphs. *jsPlumb* is not integrated in the React/Redux architecture, but it manages both visual and data aspects of the application. Fitting this component in the reactive architecture is a challenge. We will try to limit the impact of this issue, but it necessarily means that certain reducers will be dealing with mutation of the *jsPlumb* component as well as their normal job of predictably mutating the store state.

Third Party Software used

NodeJS (https://nodejs.org). *NodeJS* is a cross-platform combination of the JavaScript V8 engine from Google and a set of libraries for performing asynchronous I/O. It is used as a platform to drive server backends as well as to run command-line applications and tools, both client- and server-side. A large amount of tools for web and JavaScript development are based on NodeJS.

1. Gulp (http://gulpjs.com). A "task runner", a command-line tool that takes the specification for a set of tasks and can execute them on demand. It is built on top of NodeJS.
2. Electron Shell (http://electron.atom.io). *Electron* is a combination of NodeJS and the Chromium HTML engine. It is used to create desktop applications based on HTML5 technologies, packaging them as native executables and giving them access to Operating System resources not normally available to web-based JavaScript applications. It was created by GitHub as part of the *Atom* text editor project.
3. React (https://facebook.github.io/react/). *React* is a framework for managing and rendering HTML content inside web applications. It roughly fills the role of the "View" in the common Model-View-Controller type of architectures. React is a JS library for building user interfaces (UI).
4. Redux (https://github.com/rackt/redux). *Redux* is a predictable state container for JavaScript apps. It is an implementation of the Flux architecture described by Facebook as part of the React project. It has become the de facto standard Flux implementation for many projects due to its simplicity and flexibility.
5. Bootstrap (http://getbootstrap.com/). *Bootstrap* is a UI library of components and styles created by Twitter to facilitate the creation of responsive web projects that work well across different browsers and devices.
6. React-Bootstrap (https://react-bootstrap.github.io/). *React-Bootstrap* is a derivation of Bootstrap that implements the UI widgets and React components.
7. jsPlumb Community Edition (https://jsplumbtoolkit.com). *jsPlumb* is a library that facilitates the creation, display and editing of connected graphs of nodes in HTML.
8. Browserify (http://browserify.org). *Browserify* is a module bundler for JavaScript projects, to simplify dependency management, without using a full NodeJS implementation.
9. Babel (https://babeljs.io/). *Babel* is a compiler that transforms modern versions of the JavaScript language to older versions, compatible with more browsers and platforms.
10. jQuery (https://jquery.com). *jQuery* is a widely used library of utilities and compatibility helpers for HTML web pages and applications.

6.1.3 Evaluation and validation

Due to the nature of the tool, currently only unit tests have been performed. When a more complete version is available, with feedback from actual development work, other testing scenarios will be provided.

6.1.4 Information for developers

Documentation about the tool will be kept both in the code repository (https://github.com/GlassOceanos/nubomedia-graphedit.git), as well as in a specific doc repository (http://readthedocs.org/projects/nubomedia-graph-editor/). The latter one follows the "read the docs" style. This section summarized the details on the installation guide.

The software prerequisites to install NUBOMEDIA Visual Development Tools are the following:

- NodeJs (https://nodejs.org)
- Gulp (htttp://gulpjs.com)

NUBOMEDIA Visual Development Tools uses several JavaScript libraries, which are managed by means of NodeJs. Therefore, to get started a developer needs to first install the latest node from the OS package manager or from the NodeJS site, and then install Gulp with `npm install -g gulp`. Depending on the system, administrator privileges may be needed: `sudo npm install -g gulp`.

Next step is cloning the project where the NUBOMEDIA Visual Development Tools source code is hosted in order to build and run the editor:

```
git clone https://github.com/GlassOceanos/nubomedia-graphedit
cd nubomedia-graphedit
cd editor
npm install
```

It will take some time to download dependencies. Once it has finished, the application can be started in either of these two modes:

- `gulp electron` To run the application as standalone
- `gulp` To run the application in a web browser

`gulp` builds the application and then launches it inside a browser. `gulp electron` starts the application in standalone mode, using the Electron Shell (http://electron.atom.io). Please note that some functionality, like access to the local filesystem for saving and loading, will only be available from Electron, keeping the browser version just to help during the development process.