Kurento Room Documentation

Release 6.6.0

kurento.org

September 12, 2016

Part I

Introduction

At *Kurento* we strive to provide services for developers of advanced video applications for the Web and smartphone platforms. We found that a common use case is represented by multi-party group calls requiring a media server for advanced media processing.

This project introduces the **Rooms** API, designed for the development of conferencing applications whose centerpiece is the *Kurento Media Server*. The conference groups (rooms) will be managed using the API, which at the same time provides the means to interconnect the end-users through a Kurento Media Server instance.

The API's core module, *Room SDK*, is a Java library for the server-side and has the main functionality of managing multi-conference sessions.

Additionally, we provide *extra components* that can be used when developing applications that follow the architecture depicted above:

- **Room Server**: a container-based implementation of the server, uses *JSON-RPC* over *WebSockets* for communications with the clients
- **Room JavaScript Client**: module implementing a Room client for Web applications (works with the Room Server only)
- **Room Client**: a client library for Java web applications or Android clients (works with the Room Server only)

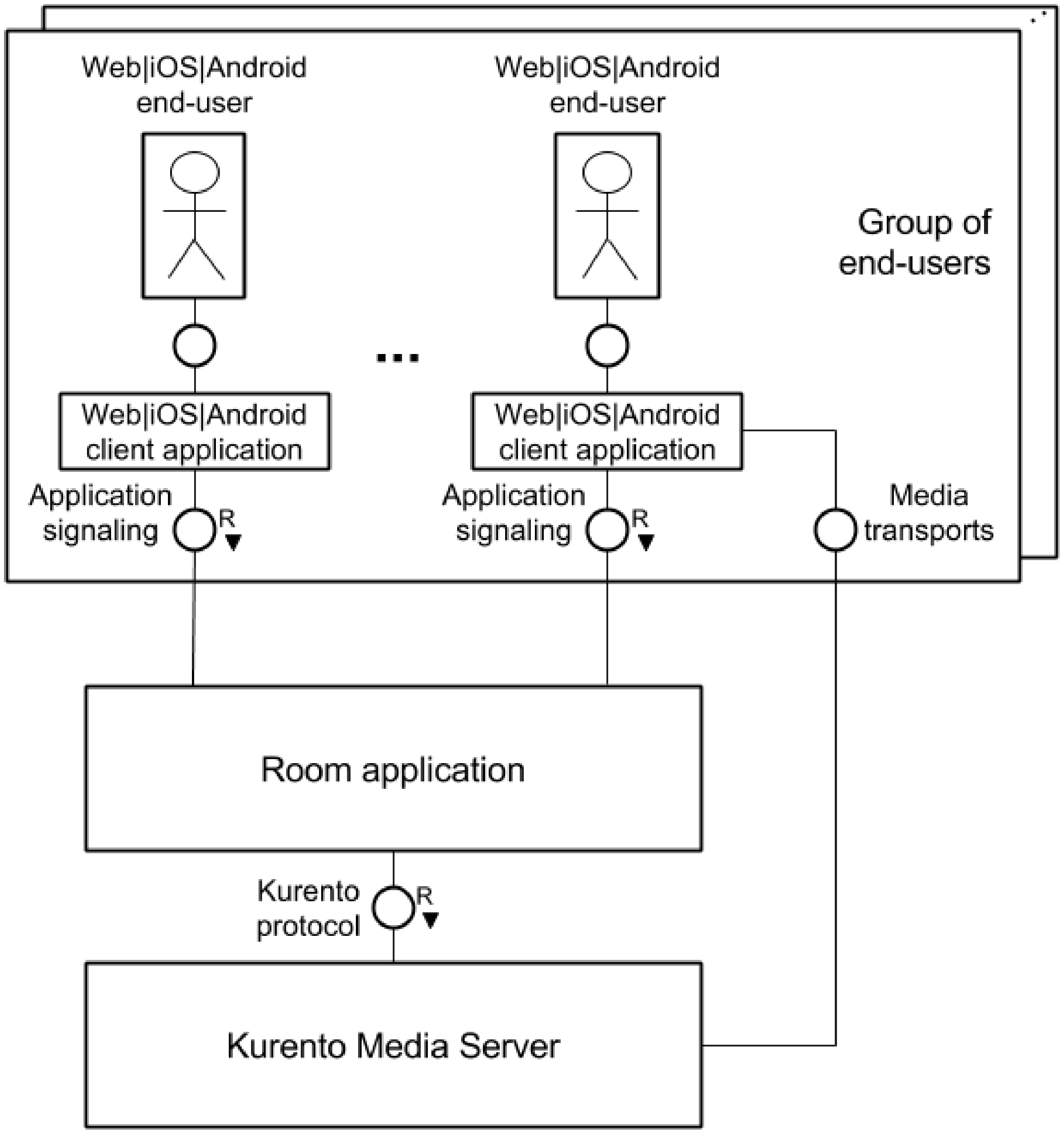

Fig. 1: *Architecture of a Room application*

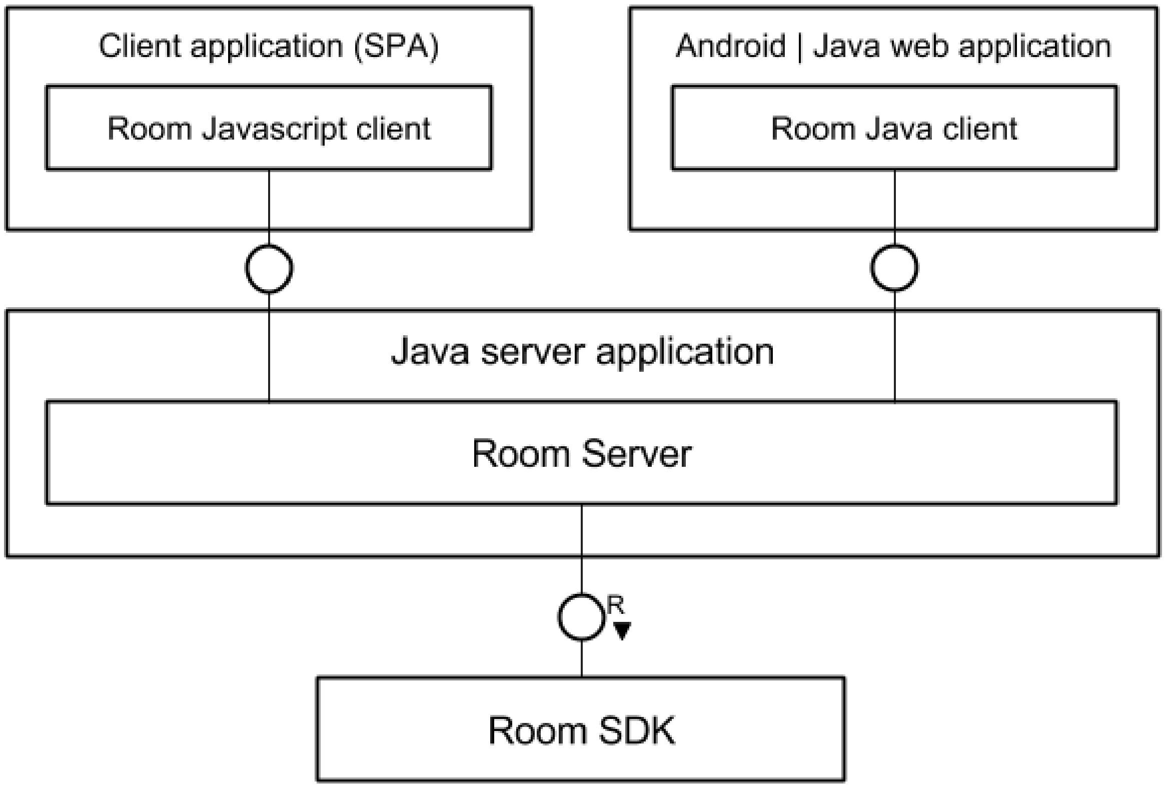

Fig. 2: *Integration of the API components*

CHAPTER 1

Core API

The core module defines a programming model for applications developed using the Java language. Some of the benefits from using this library:

- control over the lifecycle of a *multimedia* conference (**room**)
- access to typical operations required for managing the participants (join, exit, publish or receive media streams, etc.)
- simple media signaling, the application is required only to provide the media initialization and negotiation parameters received from the clients
- multimedia legs or service topologies are hidden by default (*media elements* like image filters can still be applied to a participant's stream)

This component requires access to at least one instance of a Kurento Media Server for *WebRTC* media handling.

Developers can integrate this API directly into their application, but it's important to understand that this library doesn't possess, on its own, the ability to communicate with the clients. Further details can be found in the core API section.

CHAPTER 2

Other components

Our *server* implementation, the Room Server, packs the functionality from the Room SDK with technologies such as *Spring Boot* and JSON-RPC over WebSockets. As such, it combines the client communications component and the core Room API, providing a fully-fledged Rooms server application. It can be easily integrated into other applications that use the Spring Boot framework.

Both *client* libraries are designed to be used in combination with the Room Server, as for signaling they use the JSON-RPC API exposed by our server component.

The Java client doesn't deal with media handling but only defines a Java API over the JSON-RPC protocol supported by the Room Server.

On the other hand, the JavaScript client also controls the browser's media initialization and negotiation details, enabling the developer to focus on application functionality.

At the moment, there is **no iOS client** available.

Note: Please observe that the server's WebSocket API has a limitation concerning an **user's name**, it can't contain **lower dashes** (_).

CHAPTER 3

Integration example

The **Room Demo** module is a *Single-Page Application* based on the Room Server and the Room JavaScript Client. It enables users to simultaneously establish multiple connections to other users connected to the same session or room.

Note: Even though the core module is the Room SDK, developers are free to pick whatever component they need in order to build their application: some might need some minor modifications on the client side, whilst others would want to inject their logic on server side and might even have to modify the SDK.

For example, the demo shows how to integrate some of the provided components together with the client-side technology *AngularJS* and some small modifications of the server (adds a image filter for the video streams and a custom message type).

We provide a Quick start guide for the demo's installation and execution.

There's also a short Developer guide for applications based on this API.

Part II

Quick start

For a quick initial contact with the framework, we recommend running the demo application and observing the exchange of *WebSocket* messages between the clients and the server.

Currently, the demo is only supported for Ubuntu 14.04 LTS 64bits.

CHAPTER 4

Functionalities

This project, named **kurento-room-demo**, contains the client-side implementation (HTML, JavaScript, graphic resources) of the Rooms API and embeds the room server to provide the functionality required for group communications (the so-called rooms).

Upon launch and using the *SpringBoot* framework, it starts the main application of the module **kurento-room-server** which exposes the rooms API through an WebSocket interface.

The client part has been implemented using *AngularJS* and *lumX* and it's using the room's JavaScript library for the client-side (`KurentoRoom.js`).

This application allows web clients to:

- login inside a room (creating the room if it doesn't exist)
- leave the room
- publish their media stream
- mute their video and/or audio
- enter fullscreen
- automatically subscribe to any stream published in the room and play it on the screen (video) and through the system speakers (audio)
- unsubscribe from a stream
- unpublish their media
- send messages to the other peers
- apply or remove a media filter over their published video stream (using a face overlay filter that adds a hat on top of a recognized human face)
- select which video source to display in the larger area of the browser from the available thumbnails

CHAPTER 5

Running the demo

After cloning the tutorial, it can be executed directly from the terminal by using the *Maven* `exec` plugin. To make sure the demo can be built and executed correctly, a stable release (or tag) is checked out before proceeding with the build (prevents missing dependencies, given that in *Kurento* **master** is the development branch):

```
$ git clone https://github.com/Kurento/kurento-room.git
$ cd kurento-room
# checkout the latest tag
$ git checkout $(git describe --abbrev=0 --tags)
$ cd kurento-room-demo
$ mvn compile exec:java
```

Now open the following URL in a *WebRTC*-compatible browser and connect to a new room by providing the desired user and room names: https://localhost:8443.

CHAPTER 6

Configuring the demo

There are several options or properties that might require to be modified in order for the demo to function properly.

The properties file, **kurento-room-demo.conf.json**, used in the demo's execution as described above, is located in the folder `src/main/resources` and its contents are the following:

```
{
   "kms": {
      "uris": ["ws://localhost:8888/kurento", "ws://127.0.0.1:8888/kurento"]
   },
   "app": {
      "uri": "https://localhost:8443/"
   },
   "kurento": {
      "client": {
         //milliseconds
         "requestTimeout": 20000
      }
   },
   "demo": {
      //mario-wings.png or wizard.png
      "hatUrl": "mario-wings.png",
      "hatCoords": {
         // mario-wings hat
         "offsetXPercent": -0.35F,
         "offsetYPercent": -1.2F,
         "widthPercent": 1.6F,
         "heightPercent": 1.6F

         //wizard hat
         //"offsetXPercent": -0.2F,
         //"offsetYPercent": -1.35F,
         //"widthPercent": 1.5F,
         //"heightPercent": 1.5F
      },
      "loopback" : {
         "remote": false,
         //matters only when remote is true
         "andLocal": false
      },
      "authRegex": ".*",
      "kmsLimit": 1000
   }
}
```

These properties can be overwritten on the command-line when starting the demo server:

```
$ mvn compile exec:java -Dkms.uris=[\"ws://192.168.1.99:9001/kurento\"]
```

In this example, we've instructed the demo to use a different URI of a running *KMS* instance when creating the *KurentoClient* required by the Room API.

Note: More details on the demo's configuration and execution can be found in the deployment section.

Part III

Rooms Core API

The Rooms API is based on the **Room Manager** abstraction. This manager can organize and control multi-party group calls with the aid of *Kurento* technologies.

We understand this library as an SDK for any developer that wants to implement a Room server-side application.

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.

CHAPTER 7

Understanding the API

The manager deals with two main concepts:

- **rooms** - virtual groups of peers, with the limitation that an user can be belong to only one at a time. To identify them we use their names.
- **participants** - virtual representation of a end-user. The application will provide a string representation of the user level that should suffice to uniquely identify this participant.

Given the nature of the applications using our API, it's expected that the end-users will try to connect to existing rooms (or create new ones) and publish or receive media streams from other peers.

When using this SDK, the application's job is to receive and translate *messages* from the end-users' side into *requests* for a Room Manager instance.

Some of API's methods not only deal with room management, but also with the media capabilities required by the participants. The underlying media processing is performed through a library called *Kurento Client*, which can raise events when certain conditions are met for some of the media objects created by the manager. In turn, the information gathered by handling these events is sometimes required to be notified to the end-user. The manager notifies the application of the most important events by using an interface called **Room Handler**, for which the application must provide an implementation.

We provide two types of Room Manager that expose almost the same interface (the same method names but with different signatures):

- `org.kurento.room.RoomManager`: the default implementation.
- `org.kurento.room.NotificationRoomManager`: an implementation that defines a model for sending the notifications or the responses back to the clients.

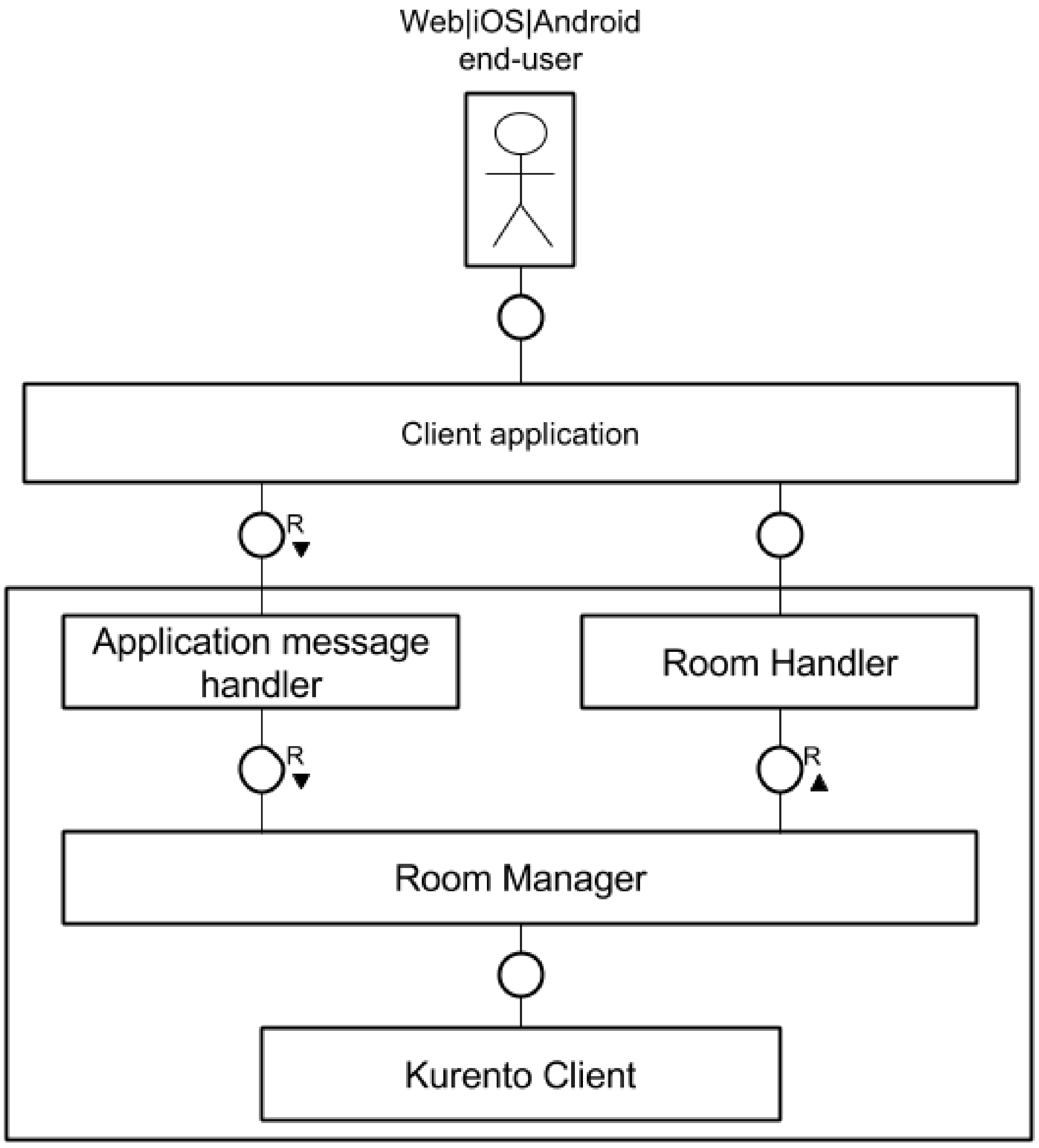

Fig. 7.1: *Room Manager integration*

CHAPTER 8

RoomManager

There are two requirements for creating a new (regular) room manager, and they are to provide implementations for:

- the Room Handler in charge of events triggered by internal media objects
- a Kurento Client Manager that will be used to obtain instances of Kurento Client

For client-originated requests, the application is required to inform the remote parties of the outcome of executing the requests, such as informing all participants in a room when one of them has requested to publish her media.

There is another type of methods that attend to application-originated requests (or business logic), in this case the application if free to interpret the result and to act upon it.

CHAPTER 9

Events - RoomHandler

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.

9.1 Room and RoomHandler relations

The following is a table detailing the server events that will resort to methods from Room Handler.

Events	**RoomHandler**
gathered ICE candidate	onSendIceCandidate
pipeline error	onPipelineError
media element error	onMediaElementError

CHAPTER 10

NotificationRoomManager

There are two requirements when instantiating a notification room manager, and they are to provide implementations for:

- a communication interface that can send messages or notifications back to the application's end users AND/OR a notification room event handler that will take the control over the notifications' lifecycle
- a Kurento Client Manager that will be used to obtain concrete instances of Kurento Client

The room event handler has been designed to provide feedback to the application with the result obtained from processing a user's request.

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:

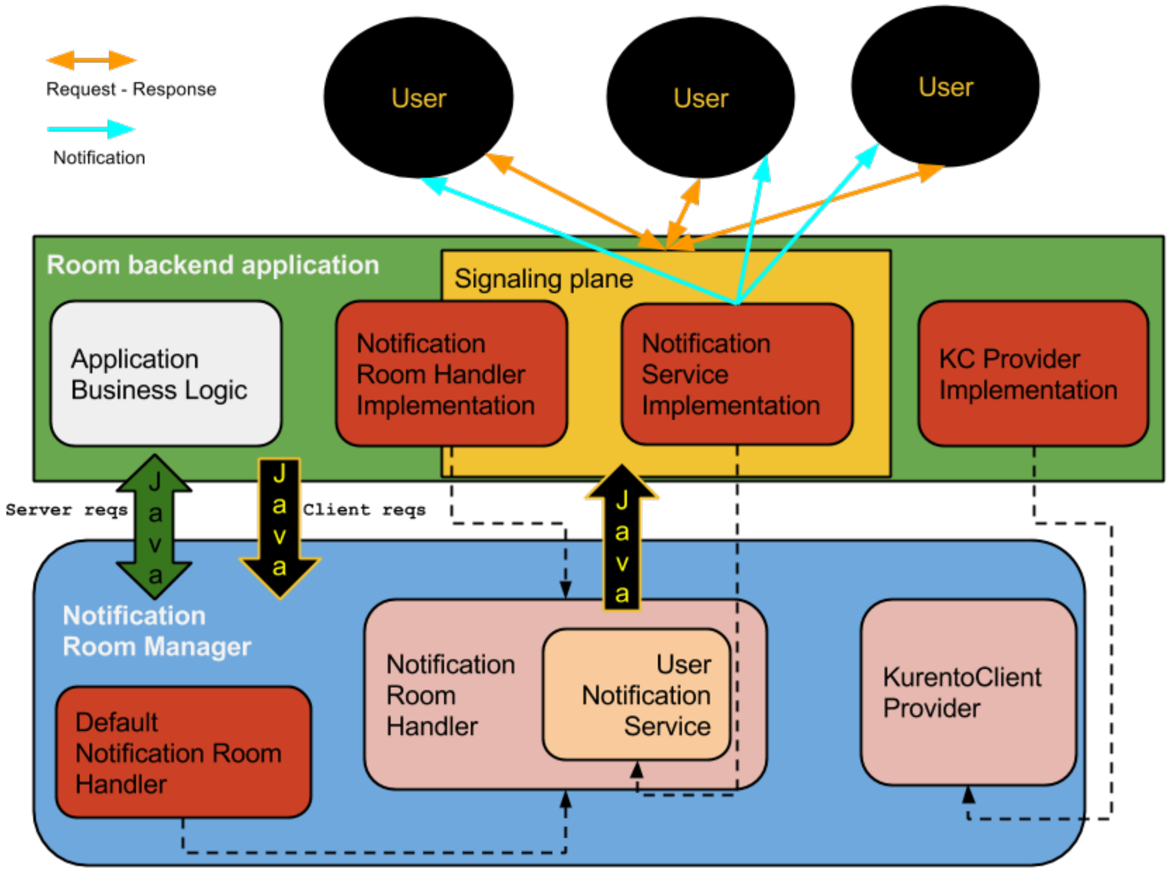
Request - Response
Notification
User
User
User
Room backend application
Signaling plane
Application Business Logic
Notification Room Handler Implementation
Notification Service Implementation
KC Provider Implementation
Server reqs
Java
Java
Client reqs
Java
Notification Room Manager
Default Notification Room Handler
Notification Room Handler
User Notification Service
KurentoClient Provider

Fig. 10.1: *Notification Room Manager*

CHAPTER 11

Notifications design - UserNotificationService

This specification was planned so that the room manager could send notifications or responses back to the remote peers whilst remaining isolated from the transport or communications layers. The notification API is used by the our implementation of the `NotificationRoomHandler` (the class `DefaultNotificationRoomHandler`).

The API's methods were defined based on the protocol *JSON-RPC* and its messages format. It is expected but not required for the client-server communications to use this protocol.

It is left for the developer to provide an implementation for this API.

If the developer chooses another mechanism to communicate with the client, they will have to use their own implementation of `NotificationRoomHandler` which will completely decouple the communication details from the room API.

Notifications design - NotificationRoomHandler

Through this interface, the room API passes the execution result of client primitives to the application and from there to the clients. It's the application's duty to respect this contract. These methods all return `void`.

Several of the methods will be invoked as a result of things happening outside of a user request scope: room closed, user evicted and the ones inherited from the `RoomHandler` interface.

12.1 NotificationRoomManager and NotificationRoomHandler relations

The following is a table detailing the methods from the `NotificationRoomManager` that will resort to methods from `NotificationRoomHandler` (also inherited methods).

NotificationRoomManager	NotificationRoomHandler
joinRoom	onParticipantJoined
leaveRoom	onParticipantLeft
publishMedia	onPublishMedia
unpublishMedia	onUnpublishMedia
subscribe	onSubscribe
unsubscribe	onUnsubscribe
sendMessage	onSendMessage
onIceCandidate	onRecvIceCandidate
close room (Server action)	onRoomClosed
evict participant (Server action)	onParticipantEvicted
gathered ICE candidate (Server event)	onSendIceCandidate
pipeline error (Server event)	onPipelineError
media element error (Server event)	onParticipantMediaError

CHAPTER 13

KurentoClientProvider

This service was designed so that the room manager could obtain a Kurento Client instance at any time, without requiring knowledge about the placement of the *KMS* instances.

It is left for the developer to provide an implementation for this interface.

CHAPTER 14

POJOs

The following classes are used in the requests and responses defined by the Rooms API.

- `UserParticipant` - links the participant's identifier with her user name and a flag telling if the user is currently streaming media.
- `ParticipantRequest` - links the participant's identifier with a request id (optional identifier of the request at the communications level, included when responding back to the client; is nullable and will be copied as is). Used in the notification variant of the **Room Manager**.
- `RoomException` - runtime exception wrapper, includes:
  - `code` - Number that indicates the error type that occurred
  - `message` - String providing a short description of the error

Part IV

Developer guide

CHAPTER 15

Quick hints

These are some of the design and architecture requirements that an application has to fulfill in order to use the Room API:

- include the SDK module to its dependencies list
- create an instance of one of the two **Room Manager** types by providing implementations for the following interfaces:
  - `RoomHandler`
  - `KurentoClientProvider`
- develop the client-side of the application for devices that support *WebRTC* (*hint:* or use our **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 (*hint:* or use the *WebSockets* API from `kurento-room-server`)
- implement a server-side handler for client messages, that will use the `RoomManager` to process these requests (*hint:* we provide a *JSON-RPC* handler in `kurento-room-server`)
- choose a response and notification mechanism for the communication with the clients (*hint:* JSON-RPC notification service from `kurento-room-server`)

About the technology stacks that can or should be used to implement a Rooms application:

- *WebSockets* as transport for messages between the server and the clients (and maybe *JSON-RPC* for the messages format).
- *Spring Boot* for the easy configuration and integration with some of Kurento's modules. It also provides a WebSockets library.

And of course, the main requirement is at least one installation of the *Kurento Media Server* that has to be accessible from the room application.

CHAPTER 16

Try the tutorial

There is a complete tutorial on how to create a multi-conference application by taking advantage of the components already provided in this project (Room SDK, Room Server and the JavaScript client). The tutorial is based on the development of the Room Demo application.

Part V

WebSocket API for Room Server

The Room Server component exposes a *WebSocket* with the default URI `wss://localhost:8443/room`, where the hostname and port depend on the current setup.

For a Room application integrating the server component, this WebSocket enables to not only receive client messages but also instantly push events to the clients, as soon as they happen.

The exchanged messages between server and clients are JSON-RPC 2.0 requests and responses. The events are sent from the server in the same way as a server's request, but without requiring a response and they don't include an identifier.

CHAPTER 17

WebSocket messages

17.1 1 - Join room

Represents a client's request to join a room. If the room does not exist, it is created. To obtain the available rooms, the client should previously use the *REST* method `getAllRooms`.

- **Method**: joinRoom
- **Parameters**:
  - **user** - user's name
  - **room** - room's name
  - **dataChannels** - optional boolean, enables *DataChannels* for the publisher
- **Example request**:

```
{"jsonrpc":"2.0","method":"joinRoom",
 "params":{"user":"USER1","room":"ROOM_1","dataChannels":true},"id":0}
```

- **Server response (result)**:
  - **sessionId** - id of the WebSocket session between the browser and the server
  - **value** - list of existing users in this room, empty when the room is a fresh one:
    - **id** - an already existing user's name
    - **streams** - list of stream identifiers that the other participant has opened to connect with the room. As only webcam is supported, will always be `[{"id":"webcam"}]`.
- **Example response**:

```
{"id":0,"result":{"value":[{"id":"USER0","streams":[{"id":"webcam"}]}],
"sessionId":"dv41ks9hj761rndhcc8nd8cj8q"},"jsonrpc":"2.0"}
```

17.2 2 - Participant joined event

Event sent by server to all other participants in the room as a result of a new user joining in.

- **Method**: participantJoined
- **Parameters**:
  - **id:** the new participant's id (username)

- **Example message**:

```
{"jsonrpc":"2.0","method":"participantJoined","params":{"id":"USER1"}}
```

17.3 3 - Publish video

Represents a client's request to start streaming her local media to anyone inside the room. The user can use the *SDP* answer from the response to display her local media after having passed through the *KMS* server (as opposed or besides using just the local stream), and thus check what other users in the room are receiving from her stream. The loopback can be enabled using the corresponding parameter.

- **Method**: publishVideo
- **Parameters**:
  - **sdpOffer**: SDP offer sent by this client
  - **doLoopback**: boolean enabling media loopback
- **Example request**:

```
{"jsonrpc":"2.0","method":"publishVideo","params":{"sdpOffer":
"v=0....apt=100\r\n"},"doLoopback":false,"id":1}
```

- **Server response (result)**
  - **sessionId:** id of the WebSocket session
  - **sdpAnswer:** SDP answer build by the the user's server WebRTC endpoint
- **Example response**:

```
{"id":1,"result":{"sdpAnswer":"v=0....apt=100\r\n",
"sessionId":"dv41ks9hj761rndhcc8nd8cj8q"},"jsonrpc":"2.0"}
```

17.4 4 - Participant published event

Event sent by server to all other participants in the room as a result of a user publishing her local media stream.

- **Method**: participantPublished
- **Parameters**:
  - **id**: publisher's username
  - **streams**: list of stream identifiers that the participant has opened to connect with the room. As only webcam is supported, will always be `[{"id":"webcam"}]`.
- **Example message**:

```
{"jsonrpc":"2.0","method":"participantPublished",
"params":{"id":"USER1","streams":[{"id":"webcam"}]}}
```

17.5 5 - Unpublish video

Represents a client's request to stop streaming her local media to her room peers.

- **Method**: unpublishVideo
- **Parameters**: No parameters required
- **Example request**:

```
{"jsonrpc":"2.0","method":"unpublishVideo","id":38}
```

- **Server response (result)**
  - **sessionId**: id of the WebSocket session
- **Example response**:

```
{"id":1,"result":{"sessionId":"dv41ks9hj761rndhcc8nd8cj8q"},"jsonrpc":"2.0"}
```

17.6 6 - Participant unpublished event

Event sent by server to all other participants in the room as a result of a user having stopped publishing her local media stream.

- **Method**: participantUnpublished
- **Parameters**:
  - **name** - publisher's username
- **Example message**:

```
{"method":"participantUnpublished","params":{"name":"USER1"}, "jsonrpc":"2.0"}
```

17.7 7 - Receive video

Represents a client's request to receive media from participants in the room that published their media. This method can also be used for loopback connections.

- **Method**: receiveVideoFrom
- **Parameters**:
  - **sender**: id of the publisher's endpoint, build by appending the publisher's name and her currently opened stream (usually webcam)
  - **sdpOffer**: SDP offer sent by this client
- **Example request**:

```
{"jsonrpc":"2.0","method":"receiveVideoFrom","params":{"sender":
"USER0_webcam","sdpOffer":"v=0....apt=100\r\n"},"id":2}
```

- **Server response (result)**
  - **sessionId**: id of the WebSocket session
  - **sdpAnswer**: SDP answer build by the other participant's WebRTC endpoint
- **Example response**:

```
{"id":2,"result":{"sdpAnswer":"v=0....apt=100\r\n", "sessionId":"dv41ks9hj761rndhcc8nd8cj8q"},"j
```

17.8 8 - Unsubscribe from video

Represents a client's request to stop receiving media from a given publisher.

- **Method**: unsubscribeFromVideo
- **Parameters**:
  - **sender**: id of the publisher's endpoint, build by appending the publisher's name and her currently opened stream (usually webcam)
- **Example request**:

```
{"jsonrpc":"2.0","method":"unsubscribeFromVideo","params":{"sender":
"USER0_webcam"},"id":67}
```

- **Server response (result)**

  "sessionId" - id of the WebSocket session

- **Example response**:

```
{"id":2,"result":{"sdpAnswer":"v=0....apt=100\r\n",
 "sessionId":"dv41ks9hj761rndhcc8nd8cj8q"},"jsonrpc":"2.0"}
```

17.9 9 - Send ICE Candidate

Request that carries info about an *ICE* candidate gathered on the client side. This information is required to implement the *trickle ICE* mechanism. Should be sent whenever an **ICECandidate** event is created by a *RTCPeerConnection*.

- **Method**: onIceCandidate
- **Parameters**:
  - **endpointName**: the name of the peer whose ICE candidate was found
  - **candidate**: the candidate attribute information
  - **sdpMLineIndex**: the index (starting at zero) of the m-line in the SDP this candidate is associated with
  - **sdpMid**: media stream identification, "audio" or "video", for the m-line this candidate is associated with
- **Example request**:

```
{"jsonrpc":"2.0","method":"onIceCandidate","params":
    {"endpointName":"USER1","candidate":
        "candidate:2023387037 1 udp 2122260223 127.0.16.1 48156 typ host generation 0",
        "sdpMid":"audio",
        "sdpMLineIndex":0
    },"id":3}
```

- **Server response (result)**:
  - **sessionId**: id of the WebSocket session
- **Example response**:

```
{"id":3,"result":{"sessionId":"dv41ks9hj761rndhcc8nd8cj8q"},"jsonrpc":"2.0"}
```

17.10 10 - Receive ICE Candidate event

Server event that carries info about an ICE candidate gathered on the server side. This information is required to implement the trickle ICE mechanism. Will be received by the client whenever a new candidate is gathered for the local peer on the server.

- **Method**: iceCandidate
- **Parameters**:
  - **endpointName**: the name of the peer whose ICE candidate was found
  - **candidate**: the candidate attribute information
  - **sdpMLineIndex**: the index (starting at zero) of the m-line in the SDP this candidate is associated with
  - **sdpMid**: media stream identification, “audio” or “video”, for the m-line this candidate is associated with
- **Example message**:

```
{"method":"iceCandidate","params":{"endpointName":"USER1",
"sdpMLineIndex":1,"sdpMid":"video","candidate":
"candidate:2 1 UDP 1677721855 127.0.1.1 58322 typ srflx raddr 172.16.181.129 rport 59597"},"json
```

17.11 11 - Leave room

Represents a client’s notification that she’s leaving the room.

- **Method**: leaveRoom
- **Parameters**: NONE
- **Example request**:

```
{"jsonrpc":"2.0","method":"leaveRoom","id":4}
```

- **Server response (result)**:
  - **sessionId**: id of the WebSocket session
- **Example response**:

```
{"id":4,"result":{"sessionId":"dv41ks9hj761rndhcc8nd8cj8q"},"jsonrpc":"2.0"}
```

17.12 12 - Participant left event

Event sent by server to all other participants in the room as a consequence of an user leaving the room.

- **Method**: participantLeft
- **Parameters**:
  - **name**: username of the participant that has disconnected
- **Example message**:

```
{"jsonrpc":"2.0","method":"participantLeft","params":{"name":"USER1"}}
```

17.13 13 - Participant evicted event

Event sent by server to a participant in the room as a consequence of a server-side action requiring the participant to leave the room.

- **Method**: participantEvicted
- **Parameters**: NONE
- **Example message**:

```
{"jsonrpc":"2.0","method":"participantLeft","params":{}}
```

17.14 14 - Send message

Used by clients to send written messages to all other participants in the room.

- **Method**: sendMessage
- **Parameters**:
  - **message**: the text message
  - **userMessage**: message originator (username)
  - **roomMessage**: room identifier (room name)
- **Example request**:

```
{"jsonrpc":"2.0","method":"sendMessage","params":{"message":"My message",
"userMessage":"USER1","roomMessage":"ROOM_1"},"id":5}
```

- **Server response (result)**:
  - **sessionId**: id of the WebSocket session
- **Example response**:

```
{"id":5,"result":{"sessionId":"dv41ks9hj761rndhcc8nd8cj8q"},"jsonrpc":"2.0"}
```

17.15 15 - Message sent event

Broadcast event that propagates a written message to all room participants.

- **Method**: sendMessage
- **Parameters**:
  - **room**: current room name
  - **name**: username of the text message source
  - **message**: the text message
- **Example message**:

```
{"method":"sendMessage","params":{"room":"ROOM_1","user":"USER1",
"message":"My message"},"jsonrpc":"2.0"}
```

17.16 16 - Media error event

Event sent by server to all participants affected by an error event intercepted on a *media pipeline* or *media element*.

- **Method**: mediaError
- **Parameters**:
  - **error**: description of the error
- **Example message**:

```
{"method":"mediaError","params":{
"error":"ERR_CODE: Pipeline generic error"},"jsonrpc":"2.0"}
```

17.17 17 - Custom request

Provides a custom envelope for requests not directly implemented by the Room server. The default server implementation of handling this call is to throw a RuntimeException. There is one implementation of this request, and it's used by the demo application to toggle the hat filter overlay.

- **Method**: customRequest
- **Parameters**: Parameters specification is left to the actual implementation
- **Example request**:

```
{"jsonrpc":"2.0","method":"customRequest","params":{...},"id":6}
```

- **Server response (result)**:
  - **sessionId**: id of the WebSocket session
  - other result parameters are not specified (left to the implementation)
- **Example response**:

```
{"id":6,"result":{"sessionId":"dv41ks9hj761rndhcc8nd8cj8q"},"jsonrpc":"2.0"}
```

Part VI

REST APIs

Apart from the *WebSocket* API, clients can also interact with the Room Server component using a more conventional *Http REST* API.

CHAPTER 18

Room Server API

The Room Server component publishes a REST service with only one primitive, that can be used to obtain the available rooms.

18.1 1 - Get all rooms

Returns a list with all the available rooms' names.

- **Request method and URL**: `GET /getAllRooms`
- **Request Content-Type**: NONE
- **Request parameters**: NONE
- **Response elements**: Returns an entity of type `application/json` including a *POJO* of type `Set<String>` with the following information:

Element	Optional	Description
roomN	Yes	Name of the N-th available room

- **Response Codes**

Code	Description
200 OK	Query successfully executed

CHAPTER 19

Room Demo API

The demo application provides an additional REST service with two primitives:

- close a given room directly from the server and evict the existing participants
- one that sends the configuration loopback parameters to the client-side

19.1 1 - Close room

Closes the room

- **Request method and URL**: `GET /close?room={roomName}`
- **Request Content-Type**: NONE
- **Request parameters**:

Element	Optional	Description
{roomName}	No	Name of the room that will be closed

- **Response elements**:

Code	Description
200 OK	Query successfully executed
404 Not found	No room exists with the provided name

19.2 2 - Get client configuration

Returns a `ClientConfig` *POJO* that can be used to configure the source for the own video (only local, remote or both).

- **Request method and URL**: `GET /getClientConfig`
- **Request Content-Type**: NONE
- **Request parameters**: NONE
- **Response elements**: Returns an entity of type `application/json` including a *POJO* of type `ClientConfig` with the following information:

Element	Optional	Description
loopbackRemote	Yes	If true, display the local video from the server loopback
loopbackAndLocal	Yes	If the other parameter is true, enables the original source as well

- **Response Codes**:

Code	Description
200 OK	Query successfully executed

Part VII

Client JavaScript API

The developer of room applications can use this API when implementing the web interface.

It is a JavaScript library build upon other public APIs like *Kurento Utils* JS, Kurento *JSON-RPC* Client JS, EventEmitter, etc. This module can be added as a *Maven* dependency to projects implementing the client-side code for web browsers that support *WebRTC*.

The library is contained by the JavaScript file `KurentoRoom.js` from the module `kurento-room-client-js`.

The main classes of this library are the following:

- **KurentoRoom**: main class that initializes the room and the local stream, also used to communicate with the server
- **KurentoRoom.Room**: the room abstraction, provides access to local and remote participants and their streams
- **KurentoRoom.Participant**: a peer (local or remote) in the room
- **KurentoRoom.Stream**: wrapper for media streams published in the room

CHAPTER 20

KurentoRoom

Example:

```
var kurento = KurentoRoom(wsUri, function (error, kurento) {...});
```

Through this initialization function, we indicate the *WebSocket* URI that will be used to send and receive messages from the server.

The result of opening the WebSocket connection is announced through a callback that is passed as parameter. The callback's signature also includes as parameter a reference to the own `KurentoRoom` object, giving access to its API when the connection was established successfully.

The interface of `KurentoRoom` includes the creation of the Room and of the local stream and also, for convenience, the following:

- Disconnect an active participant, be it remote or local media. This method allows to unsubscribe from receiving media from another peer or to end publishing the local media:

```
kurento.disconnectParticipant(stream);
```

- Close the connection to the server and release all resources:

```
kurento.close();
```

- Send messages to the other peers:

```
kurento.sendMessage(room, user, message);
```

- Send a custom request whose parameters and response handling is left to the developer. In the demo application it is used to toggle the hat filter.

```
kurento.sendCustomRequest(params, function (error, response) {...});
```

- Add additional parameters to all WebSocket requests sent to server.

```
kurento.setRpcParams(params);
```

CHAPTER 21

KurentoRoom.Room

Example:

```
var room = kurento.Room(options);
```

This constructor requires a parameter which consists of the following attributes:

- **room**: mandatory, the name of the room
- **user**: mandatory, the name of the peer inside the room
- **subscribeToStreams**: optional, can be true (default value) or false. If false, the user won't get automatic subscription to the published streams, but will have to explicitly subscribe in order to receive media.

21.1 connect() method

The room interface's main component is the connect method:

```
room.connect();
```

Instead of using a callback for dealing with the result of this operation, the client must subscribe to events emitted by the room:

21.2 room-connected event

Example:

```
room.addEventListener("room-connected", function (data) {...});
```

- **data.participants**: array of existing KurentoRoom.Participant
- **data.streams**: array of existing KurentoRoom.Stream

Emitted in case the join room operation was successful.

21.3 error-room event

Example:

```
room.addEventListener("error-room", function (data) {...});
```

- **data.error**: the error object (use data.error.message for the description)

When an error occurred when trying to register into the room.

Other events emitted during the lifecycle of the room:

21.4 room-closed event

Example:

```
room.addEventListener("room-closed", function (data) {...}
```

- **data.room**: the room's name

Emitted as a result of a server notification that the room has been forcibly closed. Receiving this event is advised to be followed by an orderly exit from the room (alert the user and close all resources associated with the room).

21.5 participant-joined event

Example:

```
room.addEventListener("participant-joined", function (data) {...});
```

- **data.participant**: a new KurentoRoom.Participant

Announces that a new peer has just joined the room.

21.6 participant-left event

Example:

```
room.addEventListener("participant-left", function (data) {...});
```

- **data.participant**: the KurentoRoom.Participant instance

Announces that a peer has left the room.

21.7 participant-evicted event

Example:

```
room.addEventListener("participant-evicted", function (data) {...});
```

- **data.localParticipant**: the local KurentoRoom.Participant instance

Announces that this peer has to leave the room as requested by the application.

21.8 participant-published event

Example:

```
room.addEventListener("participant-published", function (data) {...});
```

- **data.participant**: the KurentoRoom.Participant instance

Emitted when a publisher announces the availability of her media stream.

21.9 stream-published event

Example:

```
room.addEventListener("stream-published", function(data) {...});
```

- **data.stream**: the local KurentoRoom.Stream instance

Sent after the local media has been published to the room.

21.10 stream-subscribed event

Example:

```
room.addEventListener("stream-subscribed", function(data) {...});
```

- **data.stream**: the subscribed to KurentoRoom.Stream instance

Event that informs on the success of the subscribe operation.

21.11 stream-added event

Example:

```
room.addEventListener("stream-added", function(data) {...});
```

- **data.stream**: the new KurentoRoom.Stream instance

When the room automatically added and subscribed to a published stream.

21.12 stream-removed event

Example:

```
room.addEventListener("stream-removed", function(data) {...});
```

- **data.stream**: the disposed KurentoRoom.Stream instance

A consequence of a peer disconnecting from the room or unpublishing their media.

21.13 error-media event

Example:

```
room.addEventListener("error-media", function (data) {...});
```

- **data.error**: the error message

The server is notifying of an exception in the media server. The application should inform the user about the error and, in most cases, should proceed with an orderly exit from the room.

21.14 newMessage event

Example:

```
room.addEventListener("newMessage", function (data) {...});
```

- **data.room**: the room in which the message was sent
- **data.user**: the sender
- **data.message**: the text message

Upon reception of a message from a peer in the room (the sender is also notified using this event).

CHAPTER 22

KurentoRoom.Participant

This is more of an internal data structure (the client shouldn't create instances of this type), used to group distinct media streams from the same room peer. Currently the room server only supports one stream per user.

It is a component in the data object for several emitted room events (`room-connected`, `participant-joined`, `participant-left`, `participant-published`).

CHAPTER 23

KurentoRoom.Stream

Example:

```
var localStream = kurento.Stream(room, options);
```

The initialization of the local stream requires the following parameters:

- **room**: mandatory, the KurentoRoom.Room instance
- **options**: required object whose attributes are optional
  - **participant**: to whom belongs the stream
  - **id**: stream identifier (if null, will use the String webcam)
  - **data**: enables *DataChannels*, the application can use the *sendData()* method

23.1 init method

The stream interface's main component is the init method, which will trigger a request towards the user to grant access to the local camera and microphone:

```
localStream.init();
```

Instead of using a callback for dealing with the result of this operation, the client must subscribe to events emitted by the stream:

23.2 access-accepted event

Example:

```
localStream.addEventListener("access-accepted", function () {...});
```

Emitted in case the user grants access to the camera and microphone.

23.3 access-denied event

Example:

```
localStream.addEventListener("access-denied", function () {...});
```

Sent when the user denies access to her camera and microphone.

23.4 getID() method

The identifier of the stream, usually `webcam`.

23.5 getGlobalID() method

Calculates a global identifier by mixing the owner's id (the participant name) and the local id. E.g. `user1_webcam`.

There are several other methods exposed by the `Stream` interface, they will be described in the tutorial for making a room application.

23.6 sendData() method

If the stream is local (publishing), sends data to the server endpoint as specified by the *DataChannels* protocol.

Part VIII

Client Java API

The developer of room applications can use this API when implementing a Java or an Android client.

It is actually only a wrapper over the JSON-RPC protocol used to communicate with the Room Server.

The usefulness of this module is that it allows to create and manage room participants in a programmatic manner, or that it can be used to create an Android room client.

Please note that we haven't tested if it's actually working on the Android platform (should depend on the support for the WebSocket client implementation).

CHAPTER 24

Using the library

This client can be obtained as a Maven dependency with the following coordinates:

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

With this dependency, the developer can use the class `org.kurento.room.client.KurentoRoomClient` to create rooms or connect to existing sessions.

CHAPTER 25

Usage

To connect to a Kurento 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).

25.1 Notifications

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`

25.2 Join room

```
Map<String, List<String>> newPeers = client.joinRoom(room, username, dataChannels);
```

This method sends the `joinRoom` message and returns a list containing the existing participants and their published streams.

25.3 Leave room

```
client.leaveRoom();
```

This method sends the `leaveRoom` message.

25.4 Publish

```
String sdpAnswer = client.publishVideo(sdpOffer, false);
```

This method sends the `publishVideo` message. It returns the SDP answer from the publishing media endpoint on the server.

25.5 Unpublish

```
client.unpublishVideo();
```

This method sends the `unpublishVideo` message.

25.6 Subscribe

```
String sdpAnswer = client.receiveVideoFrom(sender, sdpOffer);
```

This method sends the `receiveVideoFrom` message. It returns the SDP answer from the subscribing media end-point on the server.

25.7 Unsubscribe

```
client.unsubscribeFromVideo(sender);
```

This method sends the `unsubscribeFromVideo` message.

25.8 Send ICE Candidate

```
client.onIceCandidate(endpointName, candidate, sdpMid, sdpMLineIndex);
```

This method sends the `onIceCandidate` message, containing a local ICE Candidate for the connection with the specified endpoint.

25.9 Send message

```
client.sendMessage(userName, roomName, message);
```

This method sends the `sendMessage` message.

Part IX

Room Demo tutorial

This tutorial is a guide for developing a multiconference application using the Room SDK. It is based on the development of the demo application found in `kurento-room-demo`, which in turn depends on the `kurento-room-sdk`, `kurento-room-server` and `kurento-room-client-js` components.

The next figure tries to explain the integration of these components and the communication channels between them.

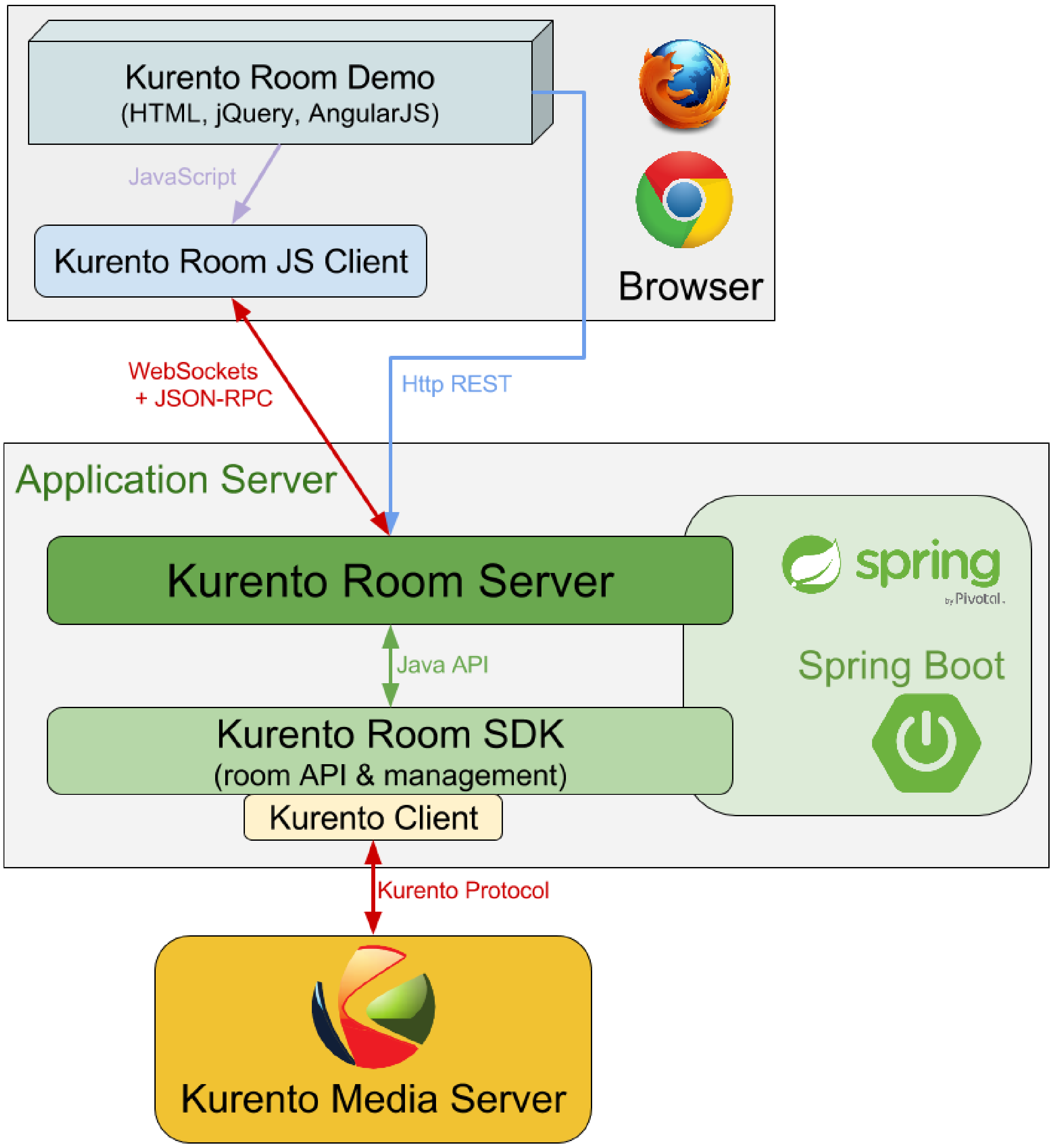

Fig. 25.1: *Kurento Room Demo integration*

CHAPTER 26

Server-side code

The main class of the room server library project is a *Spring Boot* application class, `KurentoRoomServerApp`. In this class we'll be instantiating Spring beans for the different components that make up the server-side.

Furthermore, this class with all its configuration can then be imported into application classes of other Spring projects (using Spring's `@Import` annotation or extending the server Spring Boot application class).

26.1 Room management

For managing rooms and their users, the server uses the Room SDK library. We've chosen the notification-flavored API, namely the class `NotificationRoomManager`. We have to define the manager as a Spring bean that will be injected as a dependency when needed (using the `@Autowired` annotation).

But first, we need a **UserNotificationService** implementation to provide to the `NotificationRoomManager` constructor. We'll use an instance of the type `JsonRpcNotificationService` that will store the WebSocket sessions for sending responses and notifications back to the clients.

We also require a `KurentoClientProvider` instance that we've named `KMSManager`:

```
@Bean
public NotificationRoomManager roomManager() {
    return new NotificationRoomManager(userNotificationService, kmsManager());
}
```

26.2 Signaling

For interacting with the clients, our demo application will be using the *JSON-RPC* server library developed by *Kurento*. This library is using for the transport protocol the WebSockets library provided by the Spring framework.

We register a handler for incoming JSON-RPC messages so that we can process each request depending on its method name. This handler implements the WebSocket API described earlier.

The request path is indicated when adding the handler in the method `registerJsonRpcHandlers(...)` of the `JsonRpcConfigurer` API (implemented by our Spring application class).

The handler class requires some dependencies which are passed using its constructor, the user control component and the user notifications service (these are explained below).

```
@Bean
@ConditionalOnMissingBean
public RoomJsonRpcHandler roomHandler() {
   return new RoomJsonRpcHandler(userControl(), notificationService());
}

@Override
public void registerJsonRpcHandlers(JsonRpcHandlerRegistry registry) {
   registry.addHandler(roomHandler(), "/room");
}
```

The main method of the handler, `handleRequest(...)`, will be invoked for each incoming request from the clients. All WebSocket communications with a given client will be done inside a session, for which the JSON-RPC library will provide a reference when invoking the handling method. A request-response interchange is called a transaction, also provided and from which we obtain the WebSocket session.

The application will store the session and transactions associated to each user so that our `UserNotificationService` implementation may send responses or server events back to the clients when invoked from the Room SDK library:

```
@Override
public final void handleRequest(Transaction transaction,
Request<JsonObject> request) throws Exception {
   ...
   notificationService.addTransaction(transaction, request);

   sessionId = transaction.getSession().getSessionId();
   ParticipantRequest participantRequest = new ParticipantRequest(sessionId,
   Integer.toString(request.getId()));

   ...
   transaction.startAsync();
   switch (request.getMethod()) {
     case JsonRpcProtocolElements.JOIN_ROOM_METHOD:
        userControl.joinRoom(transaction, request, participantRequest);
        break;
     ...
     default:
        log.error("Unrecognized request {}", request);
   }
}
```

26.3 Manage user requests

The handler delegates the execution of the user requests to a different component, an instance of the `JsonRpcUserControl` class. This object will extract the required parameters from the request and will invoke the necessary code from the `RoomManager`.

In the case of the `joinRoom(...)` request, it will first store the user and the room names to the session for an easier retrieval later on:

```
public void joinRoom(Transaction transaction, Request<JsonObject> request,
             ParticipantRequest participantRequest) throws ... {

   String roomName = getStringParam(request,
       JsonRpcProtocolElements.JOIN_ROOM_ROOM_PARAM);

   String userName = getStringParam(request,
       JsonRpcProtocolElements.JOIN_ROOM_USER_PARAM);
   //store info in session
   ParticipantSession participantSession = getParticipantSession(transaction);
   participantSession.setParticipantName(userName);
   participantSession.setRoomName(roomName);

   roomManager.joinRoom(userName, roomName, participantRequest);

}
```

26.4 User responses and events

As said earlier, the `NotificationRoomManager` instance is created by providing an implementation for the `UserNotificationService` API, which in this case will be an object of type `JsonRpcNotificationService`.

This class stores all opened WebSocket sessions in a map from which will obtain the Transaction object required to send back a response to a room request. For sending JSON-RPC events (notifications) to the clients it will use the functionality of the Session object.

Please observe that the notification API (`sendResponse`, `sendErrorResponse`, `sendNotification` and `closeSession`) had to be provided for the default implementation of the `NotificationRoomHandler` (included with the Room SDK library). Other variations of a room application could implement their own `NotificationRoomHandler`, thus rendering unnecessary the notification service.

In the case of sending a response to a given request, the transaction object will be used and removed from memory (a different request will mean a new transaction). Same thing happens when sending an error response:

```
@Override
public void sendResponse(ParticipantRequest participantRequest, Object result) {
   Transaction t = getAndRemoveTransaction(participantRequest);
   if (t == null) {
      log.error("No transaction found for {}, unable to send result {}",
      participantRequest, result);
      return;
   }
   try {
      t.sendResponse(result);
   } catch (Exception e) {
      log.error("Exception responding to user", e);
   }
}
```

To send a notification (or server event), we'll be using the session object. This mustn't be removed until the close session method is invoked (from the room handler, as a consequence of an user departure, or directly from the WebSocket handler, in case of connection timeouts or errors):

```
@Override
public void sendNotification(final String participantId,
   final String method, final Object params) {

   SessionWrapper sw = sessions.get(participantId);
   if (sw == null || sw.getSession() == null) {
       log.error("No session found for id {}, unable to send notification {}: {}",
          participantId, method, params);
       return;
   }
   Session s = sw.getSession();

   try {
      s.sendNotification(method, params);
   } catch (Exception e) {
      log.error("Exception sending notification to user", e);
   }
}
```

26.5 Dependencies

Kurento Spring applications are managed using *Maven*. Our server library has several explicit dependencies in its `pom.xml` file, Kurento Room SDK and Kurento JSON-RPC server are the ones used for implementing the server's functionality, while the other ones are used for testing:

```
<dependencies>
   <dependency>
      <groupId>org.kurento</groupId>
      <artifactId>kurento-room-sdk</artifactId>
   </dependency>
   <dependency>
      <groupId>org.kurento</groupId>
      <artifactId>kurento-jsonrpc-server</artifactId>
      <exclusions>
         <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>
         </exclusion>
      </exclusions>
   </dependency>
   <dependency>
      <groupId>org.kurento</groupId>
      <artifactId>kurento-room-test</artifactId>
      <scope>test</scope>
   </dependency>
   <dependency>
      <groupId>org.kurento</groupId>
      <artifactId>kurento-room-client</artifactId>
      <scope>test</scope>
   </dependency>
   <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-core</artifactId>
      <scope>test</scope>
   </dependency>
</dependencies>
```

Demo customization of the server-side

The demo adds a bit of customization to the room server by extending and replacing some of its Spring beans. All this is done in the new Spring Boot application class of the demo, `KurentoRoomDemoApp`, that extends the original application class of the server:

```
public class KurentoRoomDemoApp extends KurentoRoomServerApp {
   ...
   public static void main(String[] args) throws Exception {
      SpringApplication.run(KurentoRoomDemoApp.class, args);
   }
}
```

27.1 Custom KurentoClientProvider

As substitute for the default implementation of the provider interface we've created the class `FixedNKmsManager`, which'll allow maintaining a series of `KurentoClient`, each created from an URI specified in the demo's configuration.

27.2 Custom user control

To provide support for the additional WebSocket request type, customRequest, an extended version of `JsonRpcUserControl` was created, `DemoJsonRpcUserControl`.

This class overrides the method `customRequest(...)` to allow toggling the `FaceOverlayFilter`, which adds or removes the hat from the publisher's head. It stores the filter object as an attribute in the WebSocket session so that it'd be easier to remove it:

```
@Override
public void customRequest(Transaction transaction,
    Request<JsonObject> request, ParticipantRequest participantRequest) {

  try {
     if (request.getParams() == null
       || request.getParams().get(CUSTOM_REQUEST_HAT_PARAM) == null)
       throw new RuntimeException("Request element '" + CUSTOM_REQUEST_HAT_PARAM
           + "' is missing");

     boolean hatOn = request.getParams().get(CUSTOM_REQUEST_HAT_PARAM)
        .getAsBoolean();
        String pid = participantRequest.getParticipantId();
        if (hatOn) {
            if (transaction.getSession().getAttributes()
                .containsKey(SESSION_ATTRIBUTE_HAT_FILTER))
                throw new RuntimeException("Hat filter already on");

            log.info("Applying face overlay filter to session {}", pid);

            FaceOverlayFilter faceOverlayFilter = new FaceOverlayFilter.Builder(
            roomManager.getPipeline(pid)).build();

            faceOverlayFilter.setOverlayedImage(this.hatUrl,
                this.offsetXPercent, this.offsetYPercent, this.widthPercent,
                this.heightPercent);

            //add the filter using the RoomManager and store it in the WebSocket session
            roomManager.addMediaElement(pid, faceOverlayFilter);
            transaction.getSession().getAttributes().put(SESSION_ATTRIBUTE_HAT_FILTER,
                faceOverlayFilter);

        } else {

            if (!transaction.getSession().getAttributes()
                    .containsKey(SESSION_ATTRIBUTE_HAT_FILTER))
                throw new RuntimeException("This user has no hat filter yet");

            log.info("Removing face overlay filter from session {}", pid);

            //remove the filter from the media server and from the session
            roomManager.removeMediaElement(pid, (MediaElement)transaction.getSession()
               .getAttributes().get(SESSION_ATTRIBUTE_HAT_FILTER));

            transaction.getSession().getAttributes()
               .remove(SESSION_ATTRIBUTE_HAT_FILTER);
        }

        transaction.sendResponse(new JsonObject());

    } catch (Exception e) {
       log.error("Unable to handle custom request", e);
       try {
           transaction.sendError(e);
       } catch (IOException e1) {
           log.warn("Unable to send error response", e1);
       }
    }
}
```

27.3 Dependencies

There are several dependencies in its `pom.xml` file, Kurento Room Server, Kurento Room Client JS (for the client-side library), a Spring logging library and Kurento Room Test for the test implementation. We had to manually exclude some transitive dependencies in order to avoid conflicts:

```
<dependencies>
   <dependency>
     <groupId>org.kurento</groupId>
     <artifactId>kurento-room-server</artifactId>
     <exclusions>
        <exclusion>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-starter-logging</artifactId>
        </exclusion>
        <exclusion>
           <groupId>org.apache.commons</groupId>
           <artifactId>commons-logging</artifactId>
        </exclusion>
     </exclusions>
  </dependency>
  <dependency>
     <groupId>org.kurento</groupId>
     <artifactId>kurento-room-client-js</artifactId>
  </dependency>
  <dependency>
     <groupId>org.kurento</groupId>
     <artifactId>kurento-room-test</artifactId>
     <scope>test</scope>
  </dependency>
  <dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-log4j</artifactId>
  </dependency>
</dependencies>
```

CHAPTER 28

Client-side code

This section describes the code from the *AngularJS* application contained by `kurento-room-demo`. The Angular-specific code won't be explained, as our goal is to understand the room mechanism (the reader shouldn't worry as the indications below will also serve for a client app developed with plain or conventional JavaScript).

28.1 Libraries

Include the required JavaScript files:

```
<script src="./js/jquery-2.1.1.min.js"></script>
<script src="./js/jquery-ui.min.js"></script>
<script src="./js/adapter.js"></script>
<script src="./js/kurento-utils.js"></script>
<script src="./js/kurento-jsonrpc.js"></script>
<script src="./js/EventEmitter.js"></script>
<script src="./js/KurentoRoom.js"></script>
```

- **jQuery**: is a cross-platform JavaScript library designed to simplify the client-side scripting of HTML.
- **Adapter.js**: is a WebRTC JavaScript utility library maintained by Google that abstracts away browser differences.
- **EventEmitter**: implements an events library for the browser.
- **kurento-jsonrpc**: is a small RPC library that we'll be using for the signaling plane of this application.
- **kurento-utils**: is a Kurento utility library aimed to simplify the WebRTC management in the browser.
- **KurentoRoom**: this script is the library described earlier which is included by the `kurento-room-client-js` project.

28.2 Init resources

In order to join a room, call the initialization function from `KurentoRoom`, providing the server's URI for listening JSON-RPC requests. In this case, the room server listens for secure WebSocket connections on the request path `/room`:

```
var wsUri = 'wss://' + location.host + '/room';
```

You must also provide the room and username:

```
var kurento = KurentoRoom(wsUri, function (error, kurento) {...}
```

The callback parameter is where we'll subscribe to the events emitted by the room.

If the WebSocket initialization failed, the `error` object will not be null and we should check the server's configuration or status.

Otherwise, we're good to go and we can create a Room and the local Stream objects. Please observe that the constraints from the options passed to the local stream (audio, video, data) are being ignored at the moment:

```
room = kurento.Room({
  room: $scope.roomName,
  user: $scope.userName
});
var localStream = kurento.Stream(room, {
  audio: true,
  video: true,
  data: true
});
```

28.3 Webcam and mic access

The choice of when to join the room is left to the application, and in this one we must first obtain the access to the webcam and the microphone before calling the join method. This is done by calling the init method on the local stream:

```
localStream.init();
```

During its execution, the user will be prompted to grant access to the media resources on her system. Depending on her response, the stream object will emit the `access-accepted` or the `access-denied` event. The application has to register for these events in order to continue with the *join* operation:

```
localStream.addEventListener("access-denied", function () {
  //alert of error and go back to login page
}
```

Here, when the access is granted, we proceed with the join operation by calling connect on the room object:

```
localStream.addEventListener("access-accepted", function () {
  //register for room-emitted events
  room.connect();
}
```

28.4 Room events

As a result of the connect call, the room might emit several event types which the developer should generally be aware of.

If the connection results in a failure, the error-room event is generated:

```
room.addEventListener("error-room", function (error) {
  //alert the user and terminate
});
```

In case the connection is successful and the user is accepted as a valid peer in the room, room-connected event will be used.

The next code excerpts will contain references to the objects `ServiceRoom` and `ServiceParticipant` which are Angular services defined by the demo application. And it's worth mentioning that the `ServiceParticipant` uses streams as room participants:

```
room.addEventListener("room-connected", function (roomEvent) {

  if (displayPublished ) { //demo cofig property
    //display my video stream from the server (loopback)
    localStream.subscribeToMyRemote();
  }
  localStream.publish(); //publish my local stream

  //store a reference to the local WebRTC stream
  ServiceRoom.setLocalStream(localStream.getWebRtcPeer());

  //iterate over the streams which already exist in the room
  //and add them as participants
  var streams = roomEvent.streams;
  for (var i = 0; i < streams.length; i++) {
    ServiceParticipant.addParticipant(streams[i]);
  }
}
```

As we've just instructed our local stream to be published in the room, we should listen for the corresponding event and register our local stream as the local participant in the room. Furthermore, we've added an option to the demo to display our unchanged local video besides the video that was passed through the media server (when configured as such):

```
room.addEventListener("stream-published", function (streamEvent) {
  //register local stream as the local participant
  ServiceParticipant.addLocalParticipant(localStream);

  //also display local loopback
  if (mirrorLocal && localStream.displayMyRemote()) {
    var localVideo = kurento.Stream(room, {
      video: true,
      id: "localStream"
    });
    localVideo.mirrorLocalStream(localStream.getWrStream());
    ServiceParticipant.addLocalMirror(localVideo);
  }
});
```

In case a participant decides to publish her media, we should be aware of its stream being added to the room:

```
room.addEventListener("stream-added", function (streamEvent) {
  ServiceParticipant.addParticipant(streamEvent.stream);
});
```

The reverse mechanism must be employed when the stream is removed (when the participant leaves the room):

```
room.addEventListener("stream-removed", function (streamEvent) {
  ServiceParticipant.removeParticipantByStream(streamEvent.stream);
});
```

Another important event is the one triggered by a media error on the server-side:

```
room.addEventListener("error-media", function (msg) {
  //alert the user and terminate the room connection if deemed necessary
});
```

There are other events that are a direct consequence of a notification sent from the server, such as a room evacuation:

```
room.addEventListener("room-closed", function (msg) {
  //alert the user and terminate
});
```

Finally, the client API allows us to send text messages to the other peers in the room:

```
room.addEventListener("newMessage", function (msg) {
  ServiceParticipant.showMessage(msg.room, msg.user, msg.message);
});
```

28.5 Streams interface

After having subscribed to a new stream, the application can use one or both of these two methods from the stream interface.

stream.playOnlyVideo(parentElement, thumbnailId):

This method will append a `video` HTML tag to an existing element specified by the **parentElement** parameter (which can be either an identifier or directly the HTML tag). The video element will have autoplay on and no play controls. If the stream is local, the video will be muted.

It's expected that an element with the identifier `thumbnailId` to exist and to be selectable. This element will be displayed (`jQuery .show()` method) when a WebRTC stream can be assigned to the src attribute of the video element.

stream.playThumbnail(thumbnailId):

Creates a `div` element (class name *participant*) inside the element whose identifier is `thumbnailId`. The video from the stream is going to be played inside this `div` (*participant*) by calling `playOnlyVideo(parentElement, thumbnailId)` with it as the *parentElement*.

Using the global ID of the stream, a name tag will also be displayed onto the *participant* element as a string of text inside a div element. The style of the name tag is specified by the CSS class `name`.

The size of the thumbnail must be defined by the application. In the room demo, thumbnails start with a width of 14% which will be used until there are more than 7 publishers in the room (`7 x 14% = 98%`). From this point on, another formula will be used for calculating the width, 98% divided by the number of publishers.

Part X

Demo deployment

On machines which meet the following requirements, one can install Kurento Room applications as a system service (e.g. `kurento-room-demo`).

This section explains how to deploy (install, configure and execute) the Room Demo application. We also provide a way to run the demo without resorting to a system-wide installation.

System requirements:

- Ubuntu 14.04
- *Git* (to obtain the source code)
- Java JDK version 8
- *Maven* (for building from sources)
- *Bower* (which in turn requires *Node.js*)

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

- *Kurento Media Server* or connection with at least a running instance (to install follow the official guide)

CHAPTER 29

Installation procedures

29.1 Demo binaries

Currently, there are no binary releases of Kurento Room Demo. In order to deploy a new demo server, it is required to build it from sources.

```
$ git clone https://github.com/Kurento/kurento-room.git
$ cd kurento-room
# checkout the latest tag
$ git checkout $(git describe --abbrev=0 --tags)
```

29.2 Build from source

The demo has been configured to generate a zipped archive during the *package* phase of a Maven build. To obtain it, build the **kurento-room-demo** project together with its required modules:

```
$ cd kurento-room
$ mvn clean package -am -pl kurento-room-demo -DskipTests
```

Now unzip the generated execution binaries:

```
$ cd kurento-room-demo/target
$ unzip kurento-room-demo-6.6.0.zip
```

The directory structure of the uncompressed binaries:

- `bin/` - contains the installation and execution scripts
- `files/` - the demo's executable JAR file and other configuration files
- `sysfiles/` - used when installing as a system service

29.3 Configuration

The configuration file, `kurento-room-demo.conf.json` is located in the `files` folder, when executing the demo with normal user privileges. When installing the demo application as a system service, the configuration files will be located inside `/etc/kurento`.

```
$ cd kurento-room-demo-6.6.0
$ vim files/kurento-room-demo.conf.json
## or ##
$ vim /etc/kurento/kurento-room-demo.conf.json
```

The default content of this file:

```
{
   "kms": {
      "uris": ["ws://localhost:8888/kurento", "ws://127.0.0.1:8888/kurento"]
   },
   "app": {
      "uri": "https://localhost:8443/"
   },
   "kurento": {
      "client": {
         //milliseconds
         "requestTimeout": 20000
      }
   },
   "demo": {
      //mario-wings.png or wizard.png
      "hatUrl": "mario-wings.png",
      "hatCoords": {
         // mario-wings hat
         "offsetXPercent": -0.35F,
         "offsetYPercent": -1.2F,
         "widthPercent": 1.6F,
         "heightPercent": 1.6F

         //wizard hat
         //"offsetXPercent": -0.2F,
         //"offsetYPercent": -1.35F,
         //"widthPercent": 1.5F,
         //"heightPercent": 1.5F
      },
      "loopback" : {
         "remote": false,
         //matters only when remote is true
         "andLocal": false
      },
      "authRegex": ".*",
      "kmsLimit": 1000
   }
}
```

With the following key meanings:

- `kms.uris` is an array of WebSocket addresses used to initialize `KurentoClient` instances (each instance represents a Kurento Media Server). In the default configuration, for the same KMS the application will create two `KurentoClient` objects. The `KurentoClientProvider` implementation for this demo (`org.kurento.room.demo.FixedNKmsManager`) will return `KurentoClient` instances on a round-robin base or, if the user's name follows a certain pattern, will return the less loaded instance. The pattern check is hardcoded and SLA users are considered those whose name starts with the string special (e.g. *specialUser*).
- `kurento.client.requestTimeout` is a tweak to prevent timeouts in the KMS communications during heavy load (e.g. lots of peers). The default value of the timeout is 10 seconds.
- `app.uri` is the demo application's URL and is mainly used for building URLs of images used in media filters

(such as the hat filter). This URL must be accessible from any KMS defined in `kms.uris`.

- `demo.hatUrl` sets the image used for the `FaceOverlayFilter` applied to the streamed media when the user presses the corresponding button in the demo interface. The filename of the image is relative to the static web resources folder `img/`.
- `demo.hatCoords` represents the JSON encoding of the parameters required to configure the overlaid image. We provide the coordinates for two hat images, *mario-wings.png* and *wizard.png*.
- `demo.loopback.remote` if true, the users will see their own video using the loopbacked stream from the server. Thus, if the user enables the hat filter on her video stream, she'll be able to visualize the end result after having applied the filter.
- `demo.loopback.andLocal` if true, besides displaying the loopback media, the client interface will also provide the original (and local) media stream.
- `demo.authRegex` is the username pattern that allows the creation of a room only when it matches the pattern. This is done during the call to obtain an instance of `KurentoClient`, the provider will throw an exception if the pattern has been specified and it doesn't match the name.
- `demo.kmsLimit` is the maximum number of pipelines that can be created in a `KurentoClient`.

29.4 HTTPS

The application uses a Java keystore - `keystore.jks` - containing a self-signed certificate, which is located in the same folder as the JAR executable file.

The keystore's configuration is read from a typical `application.properties` file, read by the *Spring Boot* framework when booting up the application. Although the default name can be used during development, for installation purposes we've changed the name to `kurento-room-demo.properties`. It can be edited directly in the `files/` folder or in the service's configuration folder (`/etc/kurento`) after installing the demo.

Any changes like the keystore's name or password can be applied directly into this file.

These settings are read automatically by the application (not required to be on the command line).

```
server.port: 8443
server.address: 0.0.0.0
server.ssl.key-store: keystore.jks
server.ssl.key-store-password: kurento
server.ssl.keyStoreType: JKS
server.ssl.keyAlias: kurento-selfsigned
```

In order to disable HTTPS, remove or rename the file, or remove those lines that contain **ssl** and change the value of `server.port` to a more suitable value (recommended only if using a secure proxy with SSL).

`server.address` configures the IP address where the embedded Tomcat container binds to (default value is *0.0.0.0*, where it listens on all available addresses). It is useful when securing the application, by indicating the loopback IP and serving all connections through a secure proxy.

29.5 Logging configuration

The default logging configuration can be overwritten by editing the file `kurento-room-demo-log4j.properties`, also found in the `files` folder (or `/etc/kurento/` for system-wide installations).

```
$ cd kurento-room-demo-6.6.0
$ vim files/kurento-room-demo-log4j.properties
## or ##
$ vim /etc/kurento/kurento-room-demo-log4j.properties
```

In it, the location of the server's output log file can be set up, the default location will be `kurento-room-demo-6.6.0/logs/kurento-room-demo.log` (or `/var/log/kurento/kurento-room-demo.log` for system-wide installations).

To change it, replace the `${application.log.file}` variable with an absolute path on your system:

```
log4j.appender.file.File=${application.log.file}
# e.g. -->
log4j.appender.file.File=/home/user/demo.log
```

CHAPTER 30

Running the application

After having built and unzipped the installation files, there are two options for running the demo application server:

- **user-level execution** - doesn't need additional installation steps, can be done right away after uncompressing the installer
- **system-level execution** - requires installation of the demo application as a system service, which enables automatic startup after system reboots

In both cases, the application uses Spring Boot framework to run inside an embedded Tomcat container server, so there's no need for deployment inside an existing servlet container. If this is a requirement, modifications will have to be made to the project's build configuration (Maven) so that instead of a JAR with dependencies, the build process would generate a WAR file.

30.1 Run at user-level

After having *configured* the server instance just execute the start script:

```
$ cd kurento-room-demo-6.6.0
$ ./bin/start.sh
```

30.2 Run as daemon

First install the demo after having built and uncompressed the generated binaries. **sudo** privileges are required to install it as a service:

```
$ cd kurento-room-demo-6.6.0
$ sudo ./bin/install.sh
```

The service **kurento-room-demo** will be automatically started.

Now, you can configure the Room demo server as stated in the *previous section* and restart the service.

```
$ sudo service kurento-room-demo {start|stop|status|restart|reload}
```

30.3 Troubleshooting

For quickstarting and troubleshooting the demo use the following command to execute the *fat jar* from the **lib** folder:

```
$ cd kurento-room-demo-6.6.0/lib
$ java -jar kurento-room-demo.jar
```

30.4 Version upgrade

To update to a newer version, please repeat the installation procedures.

Part XI

Code structure

Kurento Room is hosted on github:

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

The git repository contains a Maven project with the following modules:

- kurento-room - reactor project
- kurento-room/kurento-room-sdk - module that provides a management interface for developers of multimedia conferences (rooms) applications in Java.
- kurento-room/kurento-room-server - Kurento's own implementation of a room API, it provides the WebSockets API for the communications between room clients and the server.
- kurento-room/kurento-room-client - Java library that uses WebSockets and JSON-RPC to interact with the server-side of the Room API. Can be used to implement the client-side of a room application.
- kurento-room/kurento-room-client-js - Javascript library that acts as wrapper for several JS APIs (WebRTC, WebSockets, Kurento Utils). Can be used to implement the client-side of a room application.
- kurento-room/kurento-room-demo - demonstration project, contains the client-side implementation (HTML, Javascript, *AngularJS*, *lumx*, graphic resources) and depends on the Room Server to provide the functionality required for group communications (the so-called rooms).
- kurento-room/kurento-room-basicapp - basic demonstration project, similar to `kurento-room-demo` but with a lighter client-side implementation (without any Javascript frameworks).
- kurento-room/kurento-room-test - a framework for functional tests of room applications. Required by tests from the demo and basicapp modules.

Part XII

Glossary

This is a glossary of terms that often appear in discussion about multimedia transmissions. Most of the terms are described and linked to its wikipedia, RFC or W3C relevant documents. Some of the terms are specific to *kurento*.

AngularJS Represents an open-source web application framework that tries to address many of the challenges encountered in developing single-page applications. Provides a framework for client-side model–view–controller (MVC) and model–view–viewmodel (MVVM) architectures, along with components commonly used in rich Internet applications.

See also:

AngularJS home page

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

DataChannels The WebRTC Peer-to-peer Data API lets a web application send and receive generic application data peer-to-peer. The API for sending and receiving data models the behavior of WebSockets.

See also:

http://www.html5rocks.com/en/tutorials/webrtc/datachannels/

getUserMedia The `getUserMedia()` JavaScript method is related to WebRTC because it's the gateway into that set of APIs. It provides the means to access the user's local camera/microphone stream.

See also:

`getUserMedia` from the MediaDevices interface

`getUserMedia` from the Navigator interface (*deprecated*)

Git Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

See also:

Wikipedia reference of Git

GitHub GitHub is a Web-based *Git* repository hosting service.

See also:

Wikipedia reference of GitHub

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

See also:

RFC 2616

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

See also:

RFC 5245

Wikipedia reference of ICE

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

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

Kurento Kurento is a platform for the development of multimedia enabled applications. Kurento is the Esperanto term for the English word 'stream'. We chose this name because we believe the Esperanto principles are inspiring for what the multimedia community needs: simplicity, openness and universality. Kurento is open source, released under Apache 2.0, and has several components, providing solutions to most multimedia common services requirements. Those components include: *Kurento Media Server*, *Kurento API*, *Kurento Protocol*, and *Kurento Client*.

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

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

Kurento Protocol Communication between KMS and clients by means of *JSON-RPC* messages. It is based on *WebSocket* that uses *JSON-RPC* V2.0 messages for making requests and sending responses.

Kurento Utils The Kurento Utils for Node.js and Browsers project contains a set of reusable components that have been found useful during the development of the WebRTC applications with Kurento.

See also:

GitHub repository page

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

lumx A responsive front-end framwework based on AngularJS and Google Material Design specifications. It provides a full CSS Framework built with Sass and a bunch of AngularJS components.

See also:

lumX page

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

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

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

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

See also:

Signaling Plane

Multimedia Multimedia is concerned with the computer controlled integration of text, graphics, video, animation, audio, and any other media where information can be represented, stored, transmitted and processed digitally.

There is a temporal relationship between many forms of media, for instance audio, video and animations. There 2 are forms of problems involved in

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

See also:

Wikipedia definition of

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

See also:

definition at Wikipedia

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

See also:

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

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

REST

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

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

See also:

RFC 3605

RTCPeerConnection This interface represents a WebRTC connection between the local computer and a remote peer. It is used to handle efficient streaming of data between the two peers.

RTP The is a standard packet format designed for transmitting audio and video streams on IP networks. It is used in conjunction with the *RTP Control Protocol*. Transmissions using

typically use *SDP* to describe the technical parameters of the media streams.

See also:

RFC 3550

SDP, Session Description Protocol The describes initialization parameters for a streaming media session. Both parties of a streaming media session exchange SDP files to negotiate and agree in the parameters to be used for the streaming.

See also:

RFC 4566 Definition of Session Description Protocol

RFC 4568 Security Descriptions for Media Streams in SDP

Signaling Plane It is the layer of a media system in charge of the information exchanges concerning the establishment and control of the different media circuits and the management of the network, in contrast to the transfer of media, done by the Signaling Plane.

Functions such as media negotiation, QoS parametrization, call establishment, user registration, user presence, etc. as managed in this plane.

See also:

Media Plane

SIP

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

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

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

Sphinx Documentation generation system used for Kurento projects.

See also:

Official Sphinx page

Easy and beautiful documentation with Sphinx

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

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

See also:

SRTP

SRTP

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

See also:

RFC 3711

STUN, Session Traversal Utilities for NAT STUN is a standardized set of methods to allow an end host to discover its public IP address if it is located behind a *NAT*. STUN is a client-server protocol returning the public IP address to a client together with information from which the client can infer the type of NAT it sits behind.

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

See also:

Trickle ICE IETF Draft

TLS

and its prececessor Secure Socket Layer (SSL)

See also:

RFC 5246 Version 1.2 of the Transport Layer Security protocol

TURN, Traversal Using Relays around NAT TURN is a protocol that allows for a client behind a *NAT* or firewall to receive incoming data over TCP or UDP connections. TURN places a third party server to relay messages between two clients where peer to peer media traffic is not allowed by a firewall.

User Agent Software agent that is acting on behalf of a user.

See also:

WebRTC WebRTC is an open source project that provides rich Real-Time Communcations capabilities to web browsers via Javascript and HTML5 APIs and components. These APIs are being drafted by the World Wide Web Consortium (W3C).

See also:

WebRTC Working Draft

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