JSR 309
Overview of Media Server Control API

version: Media Server Control API v1.0

Specification Leads: Tomas Ericson, Oracle, Marc Brandt, Hewlett-Packard

Please for comments go to http://jcp.org/en/jsr/detail?id=309

LICENSE

Oracle USA, Inc. and Hewlett-Packard Company ARE WILLING TO LICENSE THIS SPECIFICATION TO YOU ONLY UPON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS AGREEMENT. PLEASE READ THE TERMS AND CONDITIONS OF THIS AGREEMENT CAREFULLY. BY DOWNLOADING THIS SPECIFICATION, YOU ACCEPT THE TERMS AND CONDITIONS OF THE AGREEMENT. IF YOU ARE NOT WILLING TO BE BOUND BY IT, SELECT THE "DECLINE" BUTTON AT THE BOTTOM OF THIS PAGE.

Specification: JSR-000309 Media Server Control API ("Specification")

Version: 1.0

Status: Final Release

Specification Lead: Oracle USA, Inc. and Hewlett-Packard Company ("Specification Lead")

Release: 30-Sep-2009

Oracle USA, Inc.

500 Oracle Parkway

Redwood City, CA 94065

Hewlett-Packard Company

3000 Hanover Street

Palo Alto, CA 94304-1185 USA

LIMITED LICENSE GRANTS

1. License for Evaluation Purposes. Specification Lead hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under Specification Lead's applicable intellectual property rights to view, download, use and reproduce the Specification only for the purpose of internal evaluation. This includes (i) developing applications intended to run on an implementation of the Specification, provided that such applications do not themselves implement any portion(s) of the Specification, and (ii) discussing the Specification with any third party; and (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Specification.

2. License for the Distribution of Compliant Implementations. Specification Lead also grants you a perpetual, non-exclusive, non-transferable, worldwide, fully

paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights or, subject to the provisions of subsection 4 below, patent rights it may have covering the Specification to create and/or distribute an Independent Implementation of the Specification that: (a) fully implements the Specification including all its required interfaces and functionality; (b) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented; and (c) passes the Technology Compatibility Kit (including satisfying the requirements of the applicable TCK Users Guide) for such Specification ("Compliant Implementation"). In addition, the foregoing license is expressly conditioned on your not acting outside its scope. No license is granted hereunder for any other purpose (including, for example, modifying the Specification, other than to the extent of your fair use rights, or distributing the Specification to third parties). Also, no right, title, or interest in or to any trademarks, service marks, or trade names of Specification Lead or Specification Lead's licensors is granted hereunder. Java, and Java-related logos, marks and names are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

3. Pass-through Conditions. You need not include limitations (a)-(c) from the previous paragraph or any other particular "pass through" requirements in any license You grant concerning the use of your Independent Implementation or products derived from it. However, except with respect to Independent Implementations (and products derived from them) that satisfy limitations (a)-(c) from the previous paragraph, You may neither: (a) grant or otherwise pass through to your licensees any licenses under Specification Lead's applicable intellectual property rights; nor (b) authorize your licensees to make any claims concerning their implementation's compliance with the Specification in question.

4. Reciprocity Concerning Patent Licenses.

a. With respect to any patent claims covered by the license granted under subparagraph 2 above that would be infringed by all technically feasible implementations of the Specification, such license is conditioned upon your offering on fair, reasonable and non-discriminatory terms, to any party seeking it from You, a perpetual, non-exclusive, non-transferable, worldwide license under Your patent rights which are or would be infringed by all technically feasible implementations of the Specification to develop, distribute and use a Compliant Implementation.

b With respect to any patent claims owned by Specification Lead and covered by the license granted under subparagraph 2, whether or not their infringement can be avoided in a technically feasible manner when implementing the Specification, such license shall terminate with respect to such claims if You initiate a claim against Specification Lead that it has, in the course of performing its responsibilities as the Specification Lead, induced any other entity to infringe Your patent rights.

c Also with respect to any patent claims owned by Specification Lead and covered by the license granted under subparagraph 2 above, where the infringement of such claims can be avoided in a technically feasible manner when implementing the Specification such license, with respect to such claims, shall terminate if You initiate a claim against Specification Lead that its making, having made, using, offering to sell, selling or importing a Compliant Implementation infringes Your patent rights.

5. Definitions. For the purposes of this Agreement: "Independent Implementation" shall mean an implementation of the Specification that neither derives from any of Specification Lead's source code or binary code materials nor, except with an appropriate and separate license from Specification Lead, includes any of Specification Lead's source code or binary code materials; "Licensor Name Space" shall mean the public class or interface declarations whose names begin with "java", "javax", "com.oracle", “oracle”, “com.hp” or their equivalents in any subsequent naming convention adopted by Sun through the Java Community Process, or any recognized successors or replacements thereof; and "Technology Compatibility Kit" or "TCK" shall mean the test suite and accompanying TCK User's Guide provided by Specification Lead which corresponds to the Specification and that was available either (i) from Specification Lead's 120 days before the first release of Your Independent Implementation that allows its use for commercial purposes, or (ii) more recently than 120 days from such release but against which You elect to test Your implementation of the Specification.

This Agreement will terminate immediately without notice from Specification Lead if you breach the Agreement or act outside the scope of the licenses granted above.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS". SPECIFICATION LEAD MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT (INCLUDING AS A CONSEQUENCE OF ANY PRACTICE OR IMPLEMENTATION OF THE SPECIFICATION), OR THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE. This document does not represent any commitment to release or implement any portion of the Specification in any product. In addition, the Specification could include technical inaccuracies or typographical errors.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SPECIFICATION LEAD OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED IN ANY WAY TO YOUR HAVING, IMPELEMENTING OR OTHERWISE USING THE SPECIFICATION, EVEN IF SPECIFICATION LEAD AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend Specification Lead and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD)

acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

If you provide Specification Lead with any comments or suggestions concerning the Specification ("Feedback"), you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Specification Lead a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose.

GENERAL TERMS

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

Table of Contents

Preface

This document provides an overview of version 1.0 of the Media Server Control API for the Java Platform.

Who Should Read this Document

The intended audience for this specification includes the following:

- Developers of applications that rely on multimedia capabilities typically provided in a SIP infrastructure over a distributed Application Server and Media Server architecture.
- Implementers of this specification who want to provide control to a Media Server from an Application Server environment.

Familiarity with Media Server domain and IP multimedia (e.g. SIP, RTP) is assumed throughout.

Available documents

The following documents are available:

Document	**Filename**	**Content**
JSR 309 Overview of Media Server Control API	JSR-309-specification-overview.pdf	This document. An overview of the main Media Server Control API concepts, together with a more in-depth specification of key objects and behaviors, with examples. This document is informative.
JavaDoc: Media Server Control API for the Java Platform	mscontrol-manual.zip	The complete Javadoc and reference for the specification. This is the official complete reference for the API. In case of conflict between this document and any other document, the Javadoc is the reference.
Accompanying and documented source code samples	mscontrol-samples.zip	Code samples for simple applications that exemplify the use of the API. They are intended as an aid to understanding, but have not been rigorously tested and may not work in all situations. In particular, they are not intended to serve as a starting point for your own applications implementations.
API binary packages	mscontrol.jar	Java archive of the compiled API classes and interfaces. The Media Server Control API requires Java 2, version 1.5 or above.

Some IETF documents are referred to in this specification and provide further relevant information: on line versions of these RFCs are available at `http://www.ietf.org/rfc`

Prerequisites

In order to use the Media Server Control API, you require the following:

•Java 2, version 1.5 or above

Abbreviations

The following abbreviations are used in this document, with the meanings defined below:

AS	Application Server, typically connected to a SIP network.
MS	Media Server, typically connected to a SIP and RTP network
DTMF	Dual-tone multi-frequency
NC	`NetworkConnection`
IVR	Interactive Voice Response functions to play/record/control multimedia contents from file or streaming server
SDP	Session Description Protocol format describing media streams parameters

Typographic Conventions

Java classes, method names, parameters, literal values, and code fragments are printed in `constant width font` as are `XML elements` and documents.

Providing feedback

Please e-mail your comments to `jsr-309-comments@jcp.org`

Acknowledgements

The specification described in this document was developed under the Java Community Process 2.6 as JSR 309 http://jcp.org/en/jsr/detail?id=309 . During the whole JSR 309 duration the Expert Group has consisted of the following members (composition updated as of 2009-06-01),

listed by alphabetical order of JSR EG participating Companies, refer to EG page http://jcp.org/en/eg/view?id=309:

Eric Cheung (AT&T Research)
Ian Evans (Avaya, Inc)
Tom Strickland (Avaya, Inc)
Keith R. McFarlane (Avaya, Inc)
Chris Boulton ()
Jean-Philippe Longeray (Comverse)
Charles Hemphill (Conversational Computing Corporation)
Per Pettersson (Ericsson AB)
FAdel Al-Hezmi (Fraunhofer-Gesellschaft Institute FIRST)

Lebing Xie (Fraunhofer-Gesellschaft Institute FIRST)
Minoru Nitta (Fujitsu Limited)
Neeraj Jain (Genband, Inc)
Alain Comment (Hewlett-Packard)
Frederic Huve (Hewlett-Packard)
Scott McGlashan (Hewlett-Packard)
Jehan Monnier (Hewlett-Packard)
Mark Syrett (Hewlett-Packard)
Gary Wilson (Intervoice, Inc)
Enrico Vendruscolo (Italtel SPA)
Diego Besposvan (Mailvision LTD)
Thomas Leseney (Nexcom Systems)
Roland Auckenthaler (NMS Communications)
John Potemri (NMS)
Stephen Whynot (Nortel)
Harpreet Geekee (Nortel)
David Ferry (Open Cloud Limited)
Francois Deza (Oracle)
Stephane H. Maes (Oracle)
Eugene Ponomarenko (Oracle)
Eric Burger ()
Guillaume Teissier (Orange France SA)
Adnan Saleem (Radisys Canada Inc)
Anna Guri (Radvision Ltd.)
Ivelin Ivanov (Red Hat Middleware LLC)
Oleg Kulikov (Red Hat Middleware LLC)
Amit Bhayani (Red Hat Middleware LLC)
Venky Raju (Samsung Electronics Corporation)
Thomas Bouton (Streamwide)
Sreeram Duvur (Sun Microsystems, Inc.)
Prasad Subramanian (Sun Microsystems, Inc.)
Yoram Harel (Surf Communication Solutions Ltd)
Francesco Moggia (Telecom Italia)
Gurol Akman (Telenity)
Mehmet Ozgul (Telenity)
Wei Chen (Voxeo)
Shashi Kumar (Wipro Technologies)
Atul Varshneya

1 Overview

The JSR 309 is designed to provide server-based Java applications with multimedia capabilities. JSR 309 targets a large range of applications from simple ring-back tone applications to complex conferencing applications, by providing:

- media network connectivity to establish media streams
- IVR functions to play/record/control multimedia contents from file or streaming server
- ways to join/mix IVR function to network connection (following any topology) to create conferences and call bridges

The JSR 309 application programming interface (API) is structured to provide three sets of functionality:

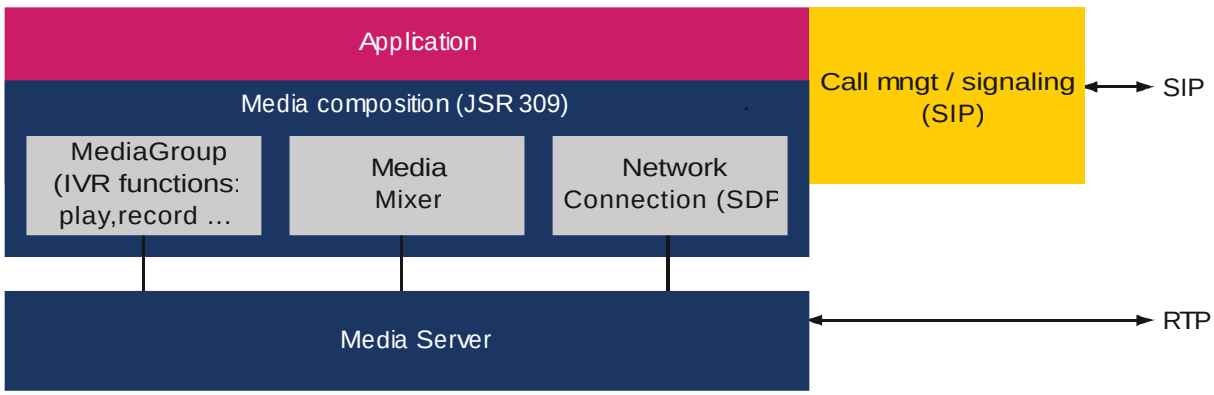


Illustration 1: JSR 309 API structure

The Network Connection (NC) object is driven by a SDP end point description, which is obtained by or given to the call-management (or signaling) part of the application. This part of the application can be implemented using the Session Initiation Protocol (SIP) Servlet API (JSR 116/289) or any API providing similar functionality. A higher level API, combining signaling and media, can be built on top of the JSR 309.

JSR 309 remains logically independent of signaling.

1.1 Goals of the Media Server Control API

The Media Server Control API is intended to provide multimedia application developers with a generic Media Server (MS) abstraction interface. It defines both a programming model and an object model for MS control independent of MS control protocols. The Media Server Control API is not an API for a specific protocol. It uses the multiple and evolving MS capabilities available in the industry today to provide an abstraction for commonly used application functions such as multi party conferencing, multimedia mixing and interaction dialogs.

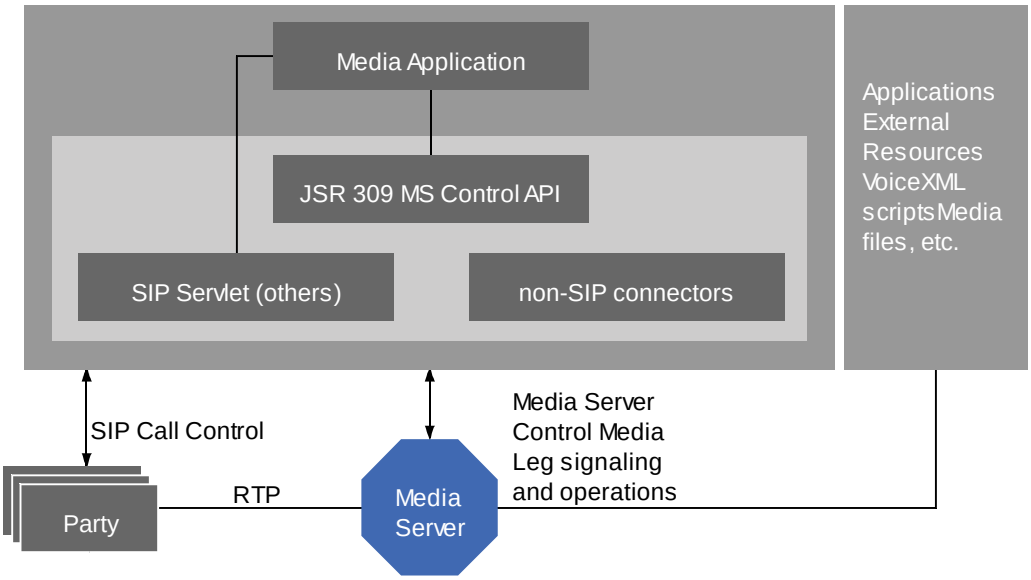


Illustration 2: JSR 309 in a SIP-based deployment architecture

Illustration 2: JSR 309 in a SIP-based deployment architecture illustrates the place of JSR 309 in a typical SIP-based deployment architecture, using a SIP Servlet.

Note: It is assumed that in general the Media Server Control API will be used in the context of a SIP-based application and a SIP-based underlying MS protocol. However the use of SIP is not a requirement and depends on the API implementation and application development choices.

1.2 MS Capabilities

The goal of the Media Server Control API is to allow applications to manipulate or combine together multimedia capabilities implemented at the MS level, in a generic manner. It is extensible to support these capabilities and also addresses the requirements of current MS's implementing features defined or required by standards organizations (e.g. IETF, 3GPP).

Such MSs typically include features such as:

•Access to multimedia stream resources and bearers and control of their characteristics, such as codecs, audio and video processing, RTP/SRTP as typical suite of protocols, other types of streams such as message-based text, mime, and so on. Call signaling control is *not* in the scope of this API.

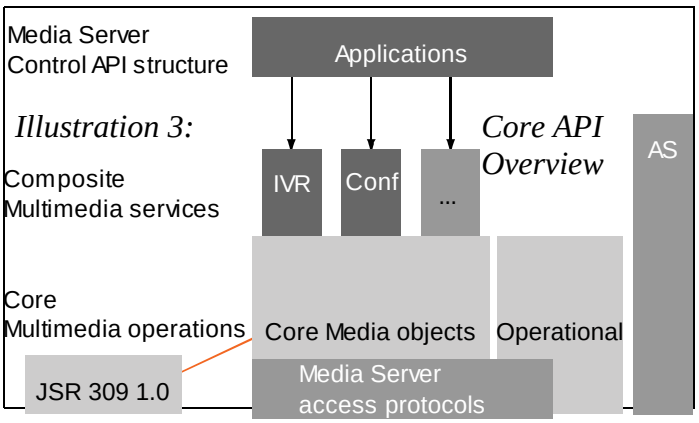
However it can provide applications with ability to handle early media capability.

- Mixing of multiple multimedia streams (for example, multimedia conference, as well as simple bridging), independently on each media type and in multiple groups and sub groups of end points (media inputs and outputs), with advanced features such as:
  - Gain control, mute and secret capability.
  - N-loudest mixing, forced mix inclusion.
  - Whispering / coaching, side-bar / split / merge mixing.
  - Active speaker notification.
  - DTMF clamping.
  - Video layout rendering with different video and audio switching capabilities.
  - Session-based messaging and mixing, future extensions to this capability are expected.
- Control of media stream resources, source or sink, to enable operations such as:
  - Play announcements, multimedia files, tones, RTSP sources.
  - Play Date / Day / Month, Time, Digits, Money, Integers and so on, for example, using speech synthesis capabilities.
  - Play multicast towards multiple media end points.
  - Record and store media stream content into specific format.
  - Play and Collect, or Play and Record information.
  - Generate or listen to messages (page-mode, session-based).
- Interaction between users, or applications, and media streams:
  - Execution of VoiceXML scripts.
  - Multimedia player or recorder control, for navigating streams (pause, resume, play at offset, channel switching).
- Processing of multimedia stream content and protocols to enable capabilities such as:
  - Detect or generate events, for example, DTMF, speech activity, floor control events.
  - Transcode, resize and adapt media streams between end points and MS such as dynamic video frame rate, bit rate or picture size adaptation per output multimedia stream.
  - Transform media: Text to Speech, Automatic Speech Recognition.
  - Mix and insert new media: audio, video, text, picture, logo, avatar or background / ambiance, RTSP sources and so on.
  - Inspect or analyze media through advanced and emerging capabilities: shape / face detection / removal, music recognition, movement / motion detection and voice transformation / masking.

1.3 Structure Overview

The API addresses two different types of application:

- Applications that need to use generic MS features in a very flexible way to control multimedia legs and multimedia service topologies. This is provided using generic **Core multimedia objects** and operation abstractions. Typical applications may involve joining resources together, playing, recording, interacting with or mixing multimedia streams.
- Applications that need a programming model to implement multimedia services that are commonly adopted or delegated to specific MS capabilities. This is provided by a composite multimedia services abstraction. Typical applications include programming of conferencing applications able to create and handle participants, and initiate standard, mute, or secret operations.



Version 1.0 of the JSR 309 Media Server Control API addresses the Core Media Objects API level. Additional composite operations can be based on the core API. This does not constrain underlying implementation choices. The definition of a *composite* API is a candidate for a future JSR.

The semantics of the Core API do not constrain it to a particular application domain and can be used generically to invoke MS capabilities, for example implemented as commands, responses and events exchanged with a specific MS implementation. The Core API mainly abstracts the MS capabilities listed in section 1.2 MS Capabilities and exposes them to applications in a generic manner through Java interfaces.

1.4 Scope of the API

The Media Server Control API does not specify the underlying implementation, like MS control protocols to be used, and does not specify a southbound driver interface to plug such protocol connectors.

Call Management is also outside the scope of Media Server Control API. Applications using the Media Server Control API are assumed to use the Application Server (AS) to provide the necessary call control APIs in order to accept incoming calls or place outgoing calls with remote user agent end points. The application is be responsible for correlating the Calls and the Media sessions it maintains with the MS. This can be done using appropriate signaling between the AS and MS over

a media server control protocol, but the definition of this mechanism is not in the scope of this API. The API is expected to hide this level of complexity for the application.

The Media Server Control API is not intended to provide detailed interfaces for Advanced Dialog management (such as controlling MRCP -Media Resource Control Protocol- resources directly from the AS). However it does provide controlling capabilities for DTMF processing and remote invocation of automatic-speech-recognition or text-to-speech as resources implemented inside the MS. In addition, the control of VoiceXML scripts running on the MS is available through an interface and is often preferred for complex dialogs.

The Media Server Control API does not force a container based implementation. However the API enables a straightforward implementation in the context of J2EE and SIP Servlet containers. Given that this API will typically be implemented in a distributed environment, some interfaces provide an asynchronous event-based version aimed at facilitating optimizations from both Application developers and Implementers, depending on the underlying MS control protocol used.

1.5 Media Server Control Core API Concepts

1.5.1 Objectives of the Core API

The main objective of the Media Server Control Core API is to provide an open, explicit and generic model to access MS capabilities that facilitate flexible programming of applications requiring a simple or advanced combination of multimedia services. Media Server Control API Applications typically *connect and process multimedia legs* among end points using multimedia functions or resources: for example, invoking multimedia processing, mixing, joining, recording of basic multimedia legs etc. The API does not attempt to restrict, nor assume, any preferred combination of joining objects and resources. This means it provides generic features and extensive controls so they can be combined in many different ways. The Core API should primarily be understood as a way of abstracting MS features.

In addition, the Media Server Control Core API provides a set of default behaviors handled by an MS that can be used when no special or advanced features are needed, for example:

- Default resource configuration descriptions for DTMF detection, video layout or other typical multimedia play and record operations. Created objects expose interfaces to the configured resources
- Encapsulation of generic player, recorder or dialog capabilities, and combinations of those. This makes it easier to control of MSs providing VoiceXML interfaces
- Runtime controls that can be processed directly at the MS level during a play or record activity for instance: stopping a recording operation when a DTMF is detected, or fast forward or backward stream navigation
- Processing of dynamic video layout descriptions that are provided to the MS

The main API interface concepts of ***Joinable*** objects `NetworkConnection`, `MediaGroup`, and `MediaMixer` have been designed to meet the above requirements. See 1.5.2 Core API and Multimedia Studio Analogy for a conceptual explanation, and 1.5.3 Media Server Control Core API Objects for an object overview.

Overall, the Core API is the result of maturing several technologies and application developments over time from a variety of experienced companies and established standards. It leverages concepts from other JSRs (for instance JTAPI Media), XML scripting architecture (e.g. VoiceXML and CCXML), and MS concepts exposed by protocols and a variety of industry efforts: SIP MSCML MSML H.248 MSCP MEDIACTRL. The Core API is designed to abstract MS concepts while remaining abstract and independent enough from any specific underlying implementation or technology choice. The API is not limited by the existing MS control protocol features but driven by Application requirements.

1.5.2 Core API and Multimedia Studio Analogy

The best way to represent the core API objects abstraction is to think of a media session with the following Multimedia Studio analogy in mind (note this is only an illustration and not restrictive to a set of features or abstract classes).

A Multimedia Studio contains many different devices, connections that can be made, and multimedia processing capabilities that can be invoked, as represented in the figure below:

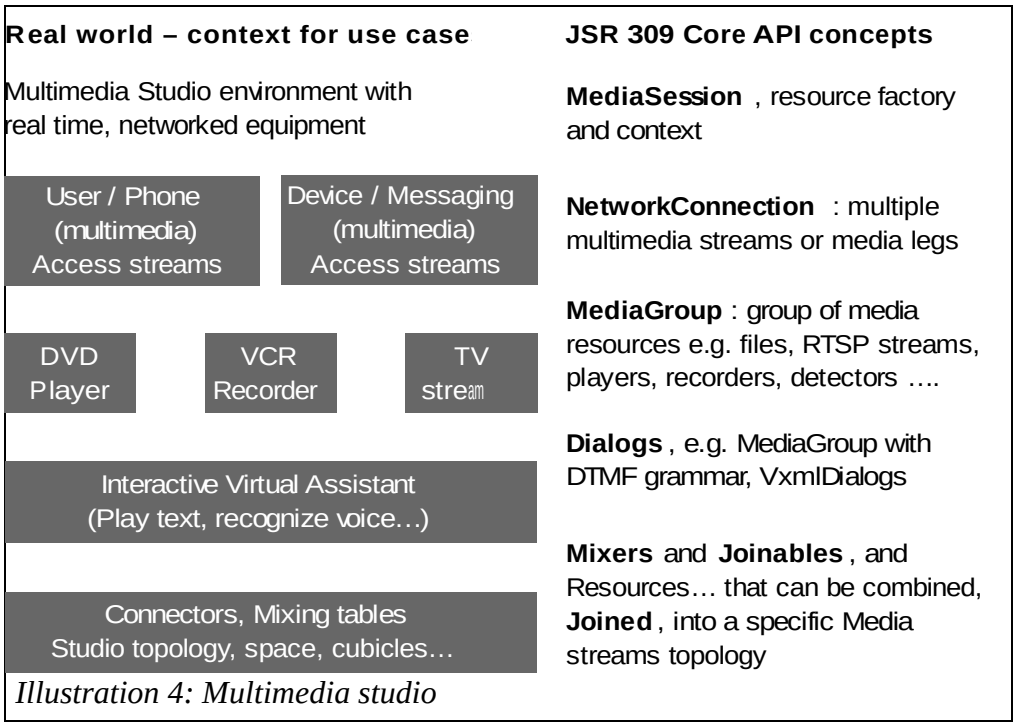


Illustration 4: Multimedia studio

Inside the Multimedia Studio, an application orchestrates the behaviors of users and devices. It might welcome participants through an interactive dialog before placing them in front of a shared DVD or TV stream with the ability to speak to each other using a conference. The application may also decide to record a specific piece of information that one user has decided to keep track of, or play music or video in the background. In other words the application is in control of the logic it uses to deliver the multimedia experience. Most importantly, the application manages the individual user's experience, for example, rendering specific video and audio based on the user's preference.

The Media Server Control API core objects can be understood with this analogy in mind. The Media Server Control API provides the foundation for implementing applications that establish an explicit connection topology and invoke available multimedia processing features based on decomposition between AS and MS. The Core API could be seen as abstracting a general-purpose multimedia studio, while the composite level abstracts a purpose-built multimedia studio, for instance a Conference room, as a specific instance of a generic multimedia studio.

The Media Server Control API describes the object interfaces made available to the application inside the Multimedia Studio. It does not describe the external network interactions. It handles

multimedia streams or legs inside the Multimedia Studio. It does not establish calls to or from the studio. It does not set any constraints on the number of MSs used to implement a studio. SDP negotiation is part of the Media Server Control API. The Core API uses it for each multimedia leg, involving the MS as it terminates the multimedia legs from the network.

The Core API can address scenarios requiring multimedia streams be connected in many different ways inside the Multimedia Studio, with any number of additional mixers, connected or isolated for some of the participants or some of the specific streams, using mute or secret modes and so on. An application using the Core API can be in control of the Multimedia Studio topology and able to **join** objects explicitly and very flexibly. There is no limitation in what object can be joined, this notion applies to every **Joinable** object.

This is not the scope of Media Server Control API to describe the external network interactions, but only to describe the object interfaces made available to the application inside the Multimedia Studio. Media Server Control API is only dealing with multimedia streams or legs inside the Multimedia Studio. It is not dealing with establishing calls to or from the studio. There is no assumption about how many MS are used to implement a studio.

1.5.3 Media Server Control Core API Objects

The Media Server Control API contains a set of core objects.

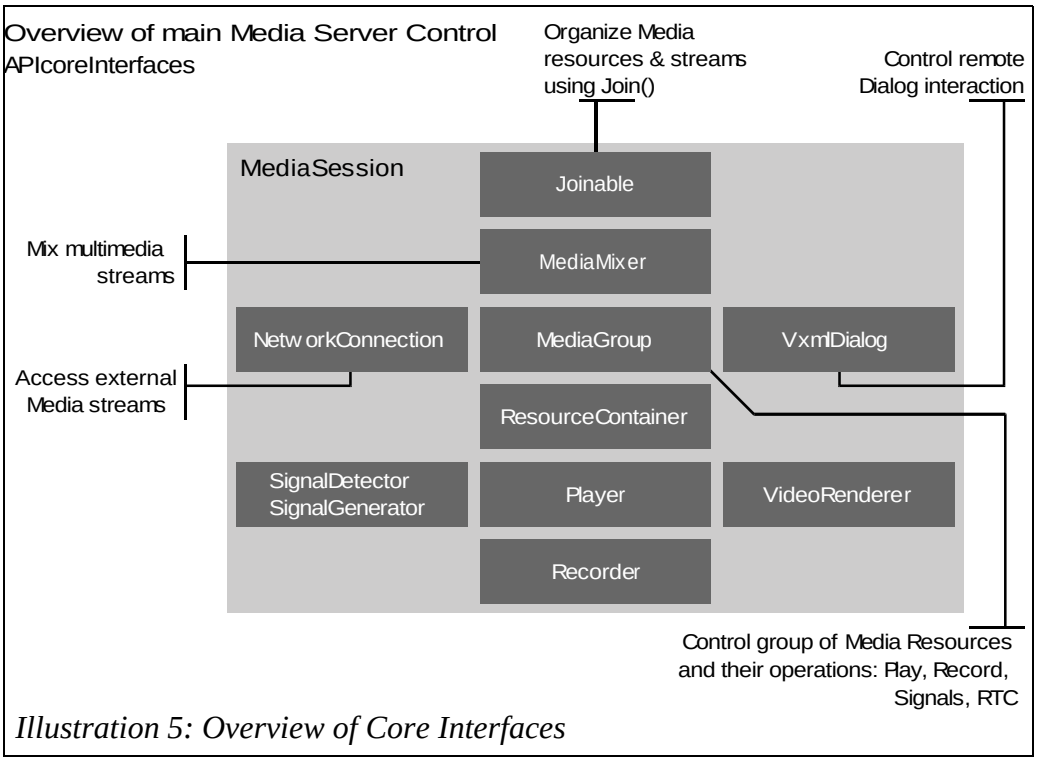


Illustration 5: Overview of Core Interfaces

`MediaSession` acts as a factory, or resource management container, for creating, configuring and releasing objects. It can be associated with a context of a specific set of media resources inside an application. From an API implementation standpoint a `MediaSession` is often associated with an instance of a MS. However the design does not restrict the way objects are be joined across

different sessions, using same or different underlying MSs.

`NetworkConnection` represents the external user agent and more specifically the multimedia leg part of it only. This object does not play a role in call control. It enables the application to reference and associate multimedia legs with specific calls, and facilitates the handling of SDP negotiation (e.g. when the MS needs to be part of the SDP negotiation).

A `NetworkConnection`'s output stream can be joined several times to other objects, that is, an outgoing stream can be split multiple times. A `MediaMixer` is required when multiple streams need to be joined to a single `NetworkConnection`'s input.

`MediaMixer` provides encapsulated resources and behaviors for managing mixing of multimedia streams, as required by Conference applications. It allows applications to connect and combine any kind and number of mixers inside an application. It has multiple inputs and can replicate its single output to be joined several times (for each stream type), like any other *Joinable* object. It handles audio / video mixing transparently with tunable mixer parameters. In a Video context, a `MediaMixer` can also contain a `VideoRenderer` allowing the application to set and customize video layouts.

`MediaGroup` represents a group of general purpose resources, for example `Player` (file, stream…) and `Recorder`, and signal detection resources reporting results through events. It can accept simultaneous and independent resource transactions like `play()`, `record()`, or `receiveSignals()`. It does not provide any specific application semantics and allows for interaction with generic multimedia features. It can be configured to be "just a player" or "just a recorder" by using or customizing default resource configurations. It can handle Runtime Controls (see RTC below) for dealing more efficiently with capabilities on the MS , such as barge-in. Runtime controls allow the MS to take actions upon events without application intervention. RTC applies during an invoked media operation.

`Joinable:` Joinable objects can be joined together, with the ability to handle individual multimedia streams or direction independently: for example, `MediaMixer.join (SEND, NetworkConnection.getJoinableStream(video))`. Output streams can be Joined to many different objects, using the split feature, (no Mixer needed). For example:

- two `NetworkConnections` can be joined together to form a bridge.
- a single `NetworkConnection` can be joined to a `MediaGroup` function to play a file or record a stream.
- three `NetworkConnections` can be joined to a `MediaMixer` to form a conference.

`Join()` topology provides explicit and dynamic operation using core API object invocation and method calls. The `Join()` concept is central to the Core API's multimedia composition capability.

`VxmDialog.` VoiceXML is one way of managing advanced Dialogs (that is dialogs not based on simple DTMF). It requires a specific Core API interface. It provides the framework for interacting with a VoiceXML dialog running on a MS in order to prepare, invoke, get events, terminate and get results from the execution of such a script. It does not define any dialog but allows invoking and interacting with the dialog execution as a core operation.

`SignalDetector` and `Generator` are typically used to handle generation or detection of DTMF, or speech.

`MediaConfig`. The Core API allows flexible creation of Core objects and controls the resources set in a specific resource container. There are several ways of doing so, for example:

- Using the default configuration defined for each Core object.
- Customizing some of the underlying resources by setting parameters.
- Providing a configuration description. In this case, the description is specific to the MS implementation and is provided as a string, for example containing XML description.

To create a `MediaGroup` configured with default player and recorder with DTMF detection resources the `PLAYER_RECORDER_SIGNALDETECTOR` predefined configuration can be used. Such predefined configurations are provided as part of Media Server Control API.

`RTC`. Run Time Controls can delegate actions to MS locally, requiring a fast reaction time, one example is forward-backward control on a stream needing fast reaction to key press. Another example is processing barge-in.

1.6 Example Use Cases

This section illustrates how the Media Server Control API Core objects can be used in two typical scenarios. Note that the objects and connections between them are not static – they evolve depending on the application's need to create and join. In the illustrations they are all shown for simplicity.

1.6.1 Video Karaoke

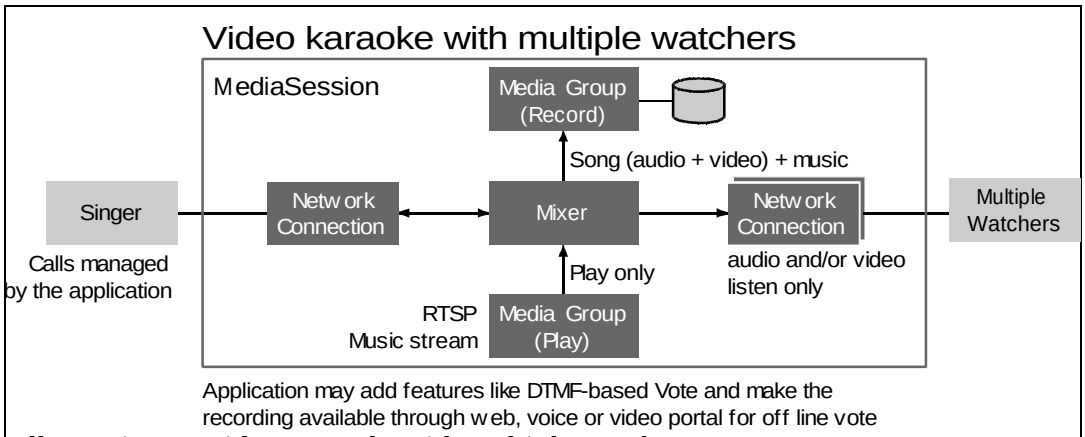


Illustration 6: Video Karaoke with multiple watchers

The karaoke example illustrates the topology for an application providing a multimedia session where a user places a video call to sing over music and be recorded. Other users can be connected and watch live, optionally vote, and replay the recording later. The application requires several object instances to be created and joined as needed. It may be necessary to tune the streams and connections depending on direction (for example, some may be in listen only mode) or available media (audio rather than video, for example).

1.6.2 Conferencing

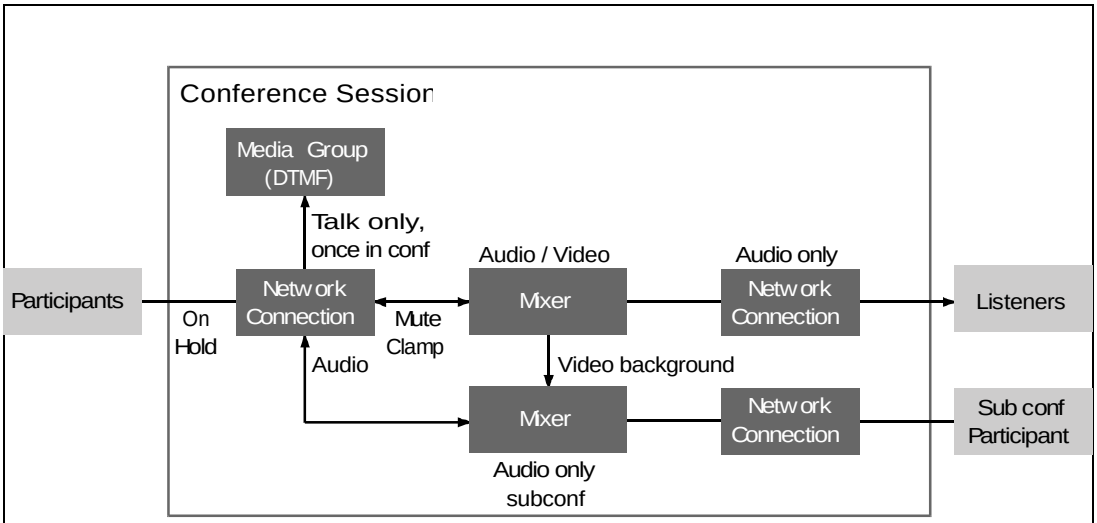


Illustration 7: Conference Session

The conferencing example shows an application that allows sub conferences to take place with some users connected as listeners only. The application can welcome users by prompting and collecting information using a DTMF `MediaGroup` before joining them to the Mixer. The main conference can be split into sub conferences which can be based on a specific media only, for example, the conference could be split up into an audio conversation with one participant while still watching the main video conference images and possibly getting background noise. The application could also provide a mute mode (listen and/or watch only) and a secret mode (cannot listen, or watch) for some events decided by the users or the application itself.

2 General Media Server Control API considerations

2.1 Application Programming Model

The Media Server Control API is designed to keep applications independent from the underlying protocol-base implementation. It also provides a consistent programming interface for media operations using different types of containers. In order to do so, the programming model supports the following mechanisms:

- An **event-**oriented programming mechanism implemented through `MediaEventNotifier` and `MediaEventListener` when applications expect *solicited or unsolicited* events or assume the completion of operations at a later time. Generally, the application assumes that initiating these operations does not block the flow unreasonably (for example, it can trigger a message sent externally but cannot block and wait for a reply). Refer to the Java Doc for the full list of event notifiers and listeners: e.g. `AllocationEventNotifier`, `JoinEventNotifier`.

- A **sequential** programming approach for regular in-line Java programming when applications expect immediate results. Generally, the application assumes that the result is accomplished upon return of the operation. In some cases it may need to block the flow, even for very short amount of time

JSR 309 defines operations that are either event-oriented due to the intrinsic nature of the media operation (e.g. a `play` or a `record`), or sequential because it is an immediate and local Java operation. There are operations that support both behaviors. In this case, implementations must support both mechanisms for portability reasons. Implementations may choose to recommend which one fits the underlying implementation capabilities best. This flexibility is provided in order to allow the application to best choose whether an immediate sequential call is best applicable or whether state-machine oriented listener is preferred. This may depend on design choices and/or container optimization.

The following illustrates API capabilities and behaviors that are built in:

- Event-oriented mechanism, enabling asynchronous implementation, must be provided and is the only mechanism for the following operations because they represent **media-dependent operations**. For that purpose the relevant media objects are also `MediaEventNotifier` thus accepting an application `MediaEventListener` to be invoked on such event:
  - Media processing activity whose duration depends on the media: `play()`, `record`(), rendering e.g. `VideoRendererEvent`, dialog control `VxmlDialogEvent`, etc.
  - Media events that are typically unsolicited: activity change, inputs/talker change, DTMF detection and generation `SignalDetectorEvent`, `receiveSignals`, `emitSignals`, typically triggered through `PlayerEvent` or `RecorderEvent`, intermediate dialog event (e.g. `VxmlDialogEvent`)

- Event-oriented mechanism must be implemented, and may be used by applications, for operations that **change the topology** thus allowing asynchronous usage of operations such as `join()` or `unjoin()` : `Joinable.joinInitiate(Direction dir, Joinable other, Serializable context)` will trigger an event when the join operation is completed. Sequential form of these operations must be implemented as well.
- Event-oriented mechanism must be implemented, and may be used by applications, for **resource reservation or modification** that enables application to create objects while relying on an asynchronous and possibly time-consuming allocation of underlying resources. Reservation of resources should apply to new or modified objects that are part of a media topology. A `confirm()` mechanism is available for such `ResourceContainer` objects and triggers an event back when allocation is completed.
- Other operations do not have an event-oriented mechanism as they are expected to return an immediate result.

2.1.1 Implementation Considerations for Synchronous or Asynchronous Behavior

Refer to 2.1 Application Programming Model for preliminary definition of the application programming models. Event-oriented and sequential mechanisms refer to the application programming level. This defines how an application behaves with regards to media operations and their typical completion, i.e. for solicited as well as unsolicited events.

On the other hand an implementation of JSR 309 may have to rely on synchronous or asynchronous mechanisms for controlling the MS. This is typically determined by the underlying containers and / or underlying MS control protocol. For example joining two objects may be based on a specific protocol exchange with acknowledgment from the MS, this is by nature an asynchronous mechanism. Similarly, once a Java object is created at the API level the underlying physical resources needed at the MS may require asynchronous allocation at some point in time.

JSR 309 implementations have to choose the best approach for supporting both event-oriented and sequential mechanisms when they are defined and required by the API. In general, implementations should be asynchronous / non-blocking for all event-oriented operations and can be synchronous for sequential operations. However a mix may be required depending on the exact AS-MS communication capability or protocol. Therefore, it is possible that:

- A synchronous implementation must implement sequential operations natively and emulate event-oriented mechanisms by adding an emulation of asynchronous mechanism (e.g. with a timer) so that an event is triggered even if the operation has successfully been carried out immediately.
- An asynchronous implementation must implement an event-oriented operation natively, for example, by relying on underlying asynchronous MS events (e.g. protocol events), and must emulate sequential operations by blocking the application calling thread.

Most implementations will be a mix of this as the underlying protocol will dictate how operations can be implemented, and some operations will be natively implemented through their programming models while others would have to emulate it.

This flexibility allows JSR 309 applications to use the most appropriate programming model for

their own needs and design, and also JSR 309 to be implemented over a variety of current distribution mechanisms.

2.2 Serializability and Distributed Environment

Serialization applies only to distributable environments.

The `setAttribute` and `addListener` methods must throw an `IllegalArgument` exception if an object not implementing `Serializable` is passed to these methods.

When using `EventListeners`, the application must serialize access to media sessions and related objects, to take account of attempts by application threads to access the media session concurrently with an even listener invocation. This applies both to access by the driver invoking event listeners (call-backs) and to access by any application thread.

2.3 Thread safety

The JSR 309 implementation must be thread-safe when using common singleton resources such as factories. These resources are usually shared by several application threads and applications. This means several concurrent threads can safely access resources like a `MsControlFactory`.

Optionally, a driver can provide thread-safe implementations of `MediaSession` and for objects contained in instances of `MediaSession`. This means implementations using `MediaSession` do not have to support concurrent access by java synchronization. In this case, the application must implement appropriate locking to serialize access to instances of `MediaSession` and related objects. The locking granularity is application specific. The programmer must be aware of risks of deadlocks and inconsistency due to concurrent access in the case of several related media sessions – for example, if each participant in a conference possess their own media session.

2.4 Object creation, resource allocation and release

Object creation methods (for example, `MediaSession.createNetworkConnection()`) create local Java objects, but do not need to interact with the MS until the created object is joined or resources get allocated, so there may be no visible effect until this point. Media reservation operations can be delayed until the object is joined. Therefore, creation of objects such as `MediaGroup`, `MediaMixer`, `MixerAdapter`, `NetworkConnection`, and `VxmlDialog` immediately returns the created Java Objects.

The `confirm()` mechanism handles reservations and modifications at `ResourceContainer` level. This lets you keep creation of the API Java objects separate from resource reservation or modification.

The `ResourceContainer` is also an `AllocationEventNotifier` so that its listener can capture events about the `ResourceContainer` lifecycle. Invoking `confirm()` explicitly triggers completion of reservation or any other changes that impact resources in the container (for example, changing parameters that affect the resources allocations) and sends an event.

For this reason, creation of local Java objects requires a sequential programming approach, but objects should not block until really needed for interacting with the MS. This allows newly created objects to handle resource allocation events immediately after their creation.

Overall, the following behaviors define how objects are created:

- Creating a `NetworkConnection`. the main part of the creation lifecycle consists of setting,

or negotiating, the media streams parameters with the MS. This is typically done using the SDP mechanism. For this reason the `SdpPortManager` is event-oriented. It also supports events like `UNSOLICITED_OFFER_GENERATED` to enable the MS to report needed changes on the Media ports allocations. The `confirm()` reports on the allocation state of the `NetworkConnection` as a resource container.

- Creating a `MediaGroup` does not usually invoke actions on the MS until the relevant media operations are invoked by the embedded `Player` or `Recorder` resources. However use of confirm mechanism supports this possibility if required.
- Creating a `MediaMixer`, or `MixerAdapter` is likely to require resource reservation, for instance based on the parameters such as `MAX_PORTS`. Invoking `MediaMixer.confirm()` ensures that resource reservation and allocation is completed, especially important in cases where further media operations depend on successful resource reservation.
- `CreateVxmlDialog` is not created in the same way as a `ResourceContainer`. A VxmlDialog does not contain resources, but offers event-oriented operations to control the Dialog.

Applications need to handle resource reservation in an explicit manner. By default, resource reservation is implicit, and is usually enforced by the API implementation on the first "join" or media operation invoked on the object. However an application may require a more explicit approach to complete resource reservation from its own logic standpoint and to ensure an event is received upon completion of reservation or modification of reservation, rather than delegating it to the Media Server. In that case it must call `confirm()`. From an application programming standpoint, it can adopt one of the following approaches:

- If it does not need to confirm reservation immediately, it assumes `ResourceContainer<T>` resources will be allocated on first usage: for example on next `join()` (or `joinInitiate`) or `play()`,… or on a future call to `confirm()`.
- If it needs to synchronize its own logic with confirmation of allocation, it must explicitly set a listener `AllocationEventListener` after the call to create a specific container and call `confirm()` to wait for resources to be allocated before doing any kind of operation.

A typical application logic follows this pattern:

- `createMediaMixer(...)`
- `setListener`
- `setParameter`(….)
- `confirm`()
- `receive` `ALLOCATION_CONFIRMED`
- `join`(), or `joinInitiate`()
- etc…
- `release`()

2.4.1 Releasing Objects and Resources

From an application standpoint, once `MediaObject.release()` has been invoked, further use of released objects is forbidden. The consequences from an MS standpoint are implementation-dependent, and affect actions such as de-allocating resources, or caching resources for future reuse.

The release method can typically be implemented synchronously without blocking. If MS interaction is required to release certain resources, this should be transparent to the application. Hence the `release()` is not provided with an event-oriented model.

Release() is expected to abort pending events that have not been already delivered to listeners.

2.5 Group of Operations / Transactions

The event-oriented mechanism of the `JoinInitiate()` and allocation of resources using `confirm()` lets implementations support an asynchronous approach. Scenarios can be optimized by initiating several join operations in parallel, without having to wait for the completion of one `Join()` operation before starting the next one. This minimizes the likelihood and impact of transient states, and allows implementations to group operations in batches, if the MS protocol supports such messages.

There is no explicit mechanism visible to the application for grouping or batching operations.

2.6 Prompt and collect

This operation is considered generic enough to be included in the Core API level. This capability allows a prompt to be played before recording or collecting digits using a single core media operation invoked from the application (i.e. record with prompt, receive signals with prompt). This allows applications to perform one 'atomic' operation that can be optimized on existing protocols / MS implementations. The `MediaGroup` is especially designed to support these combined operations between its underlying resources.

This extends the capability of the `MediaGroup` (like `START_BEEP`), for typical configurations such as `PLAYER_RECORDER_SIGNALDETECTOR` or `PLAYER_SIGNALDETECTOR`, by allowing application to invoke the `Recorder.record()` or `SignalDetector.receiveSignals` and specifying an optional `PROMPT` parameter to be played before the actual record starts (`Recorder`) or before the digit(s) are collected (`SignalDetector`).

This parameter can be a URI or an array of URIs that can be accepted by the `Player` of the same `MediaGroup`. This parameter can either be positioned by default once for all for the `MediaGroup`, or can be passed on every call to a the media operation (using Parameters).

The application then receives the relevant events from the `record()` or `receiveSignal()` operations as usual.

The `PROMPT` parameter can be positioned for the whole duration of the `MediaGroup`, or on each call, following the parameter feature, and overloading rule.

It is expected that playing the prompt is delegated to the MS. If application wants tight control of the play, it needs to call `play` before the `record` or `receiveSignals` operations.

2.7 Mandatory and Optional Features

•The Media Server Control API is flexible and allows implementations to differentiate when they need to based on the effective Media Server capabilities used. This is because Media Servers are very diverse in capabilities and all features cannot be mandatory. Some Media Servers are specialized for Audio only, some for Conferencing as an example. Therefore some JSR 309 features are optional, and not required to be supported by implementations. However an implementation must implement every method of the JSR 309 API, but an implementation may choose to throw `NotSupportedException` etc for optional methods that the Media Server does not support. An implementation must also recognize every field (parameter) and provide the list of supported features through the `getSupportedFeatures() described below.`

Refer to the TCK coverage documentation that provides the complete list of mandatory / optional features.

Overview of Mandatory features:

• All core Interfaces must be implemented: `MediaGroup`, `NetworkConnection` and `MediaMixer, w`ith their underlying resources operations (like `play, record`) on `Player, Recorder, SignalDetector`. That does not mandate the support of every possible type of media content to be played at the MS level. The following `MediaGroup` capabilities must be supported:

  • basic set of predefined `MediaConfig: PLAYER, PLAYER_RECORDER_SIGNALDETECTOR, PLAYER_SIGNALDETECTOR`

  • Prompt and record, Prompt and collect using `PROMPT`

  • file:// and http:// protocols for play and record

  • `Audio` .

• `Join()` operations used to cover IVR and Conferencing use cases

• `Parameters and SupportedFeatures`. There is a mandatory mechanism (`SupportedFeatures`) described below to report if a specific MS configuration supports certain `Parameters`. As a lot of `Parameters` correspond to optional features that may not be supported by the MS but the query is mandatory.

Overview of Optional features:

•Video support:

  •`MediaConfig` with video support for `MediaGroup`, `NetworkConnection`;

  •video mixers;

  •video mixers with explicit layout: `VideoLayout, VideoRenderer`.

•Support of other play/record protocols: e.g. rtsp:// protocol for play. This remains transparent to the play operation interface, and it is up to the implementation to return appropriate operation results

•`SignalGenerator, VxmlDialog`

•Some join topologies. The API does not mandate which join topologies are supported, as many combinations are allowed. Implementations shall report the corresponding event or exception when

that applies

- joining any couple of objects
- multiple joins to a single object
- join video separately from audio
- Joining objects, or MediaSessions, between different MS: refer to Join() JavaDoc for list of mandatory and optional join topologies

Many capabilities that a MS offers are available through Parameters that must be recognized by the API, but the resulting behavior may not be supported (e.g. Video conferencing). A discovery mechanism for supported features simplifies validation of available resources.

The discovery can be performed from the `MsControlFactory`, without actually creating the objects that are required when setting a phone call. This enables the application to discover the supported features in its initialization phase, and be ready when calls arrive.

The discovery is based on the `MediaConfig` interface, and is two-fold:

1. `MsControlFactory.getMediaConfig(Configuration configuration)` enables the application to browse the available `MediaConfig`, describing the features of the `NetworkConnection`, `MediaGroup` and `MediaMixer`.
2. For each `MediaConfig`, the query `MediaConfig.getSupportedFeatures()`, returns the Sets of supported features.

Example:

```
•MediaConfig ivrConfig =
MsControlFactory.getMediaConfig(MediaGroup.PLAYER_RECORDER_SIGNALDETECTOR);
```

If the above does not throw an Exception, then the implementation supports playing, recording and detecting DTMFs.

```
•Set<Parameter> suppParams =
ivrConfig.getSupportedFeatures().getSupportedParameters();
```

If the returned set of Parameters include `SpeechDetectorConstants.INITIAL_TIMEOUT`, this indicates support for canceling a record operation if the caller does not speak (`initsilence` in MSCML, or `prespeech` in MSML).

2.8 Resource Extensibility

The `MediaConfig` mechanism defines the association of resources inside objects, such as `MediaGroup`. This mechanism can accept MS-specific descriptions as Strings.

2.9 Resource Customization

Various levels of customization of the `MediaGroup` are possible:

- •Customize a few parameters of some `Resource(s)` during a `play()` or `record()`
- •Customize an instance of `MediaGroup`,
- •Customize the default parameter values of a Resource
- •Create (clone) a new configuration of existing Resources (add, remove, change order)

3 Media composition: Basic IVR, bridging and mixing API

3.1 Introduction

There are three types of object related to media processing:

- A Network Connection (NC) represents a telephone user. It is connected to the telephony media network.

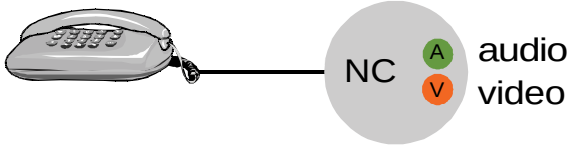
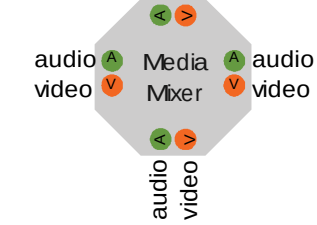


Illustration 8: Network Connection

- A MediaGroup provides IVR functions: it can play prompts and/or record messages, and detect dual tone multi frequencies (DTMFs).

Illustration 9: Media Group

- A MediaMixer has multiple inputs and outputs:



Each of these objects has its own lifecycle:

•NCs are created by the application for each incoming or outgoing call.

•MediaGroups and MediaMixers are created on the application's request, using the Media Session as a factory (using the command `createMediaGroup`, etc*)*.

An application can connect or join these objects in many different ways. All combinations are supported by the API, which makes it one of the most powerful platforms for developing advanced telephony/multimedia services, using only three types of objects and one method to connect them. Those concepts are based on the CCXML W3C specification (www.w3.org/TR/ccxml).

Some API implementations may rely on protocols that cannot make use of the full flexibility of the API, and which limit the geometry. For example, this would be the case if an MSCML-based connector provides the connectivity corresponding to the usual IVR and conference services. In this case, attempting to build more specific connections generates exceptions.

Note that 'loop', 'bridge' and 'conference' are terms with specific meanings in the context of a telephony service:

- Loop implies a single party
- Bridge is a relationship between two parties
- Conference usually implies more than two parties.

A conference with two parties, may also be considered a bridge.

The examples in the following sections illustrate the behavior and uses of JSR 309's `NetworkConnections`, `MediaGroups` and `MediaMixers`, connected using the `join` primitive.

They define a lower level of abstraction than 'bridge' or 'conference'. The examples also show the associated telephony service semantics.

3.2 Joining Media Objects

The simplest way to join media objects is to play a prompt to the user, and/or record the user's message.

It requires joining the `NetworkConnection` to a `MediaGroup`, using the `join` method:

```
NetworkConnection.join(DUPLEX, MediaGroup);
```

The effect of the **join** call is indicated by the arrows:

This process is also referred to as 'looping' a MediaGroup to a NetworkConnection.

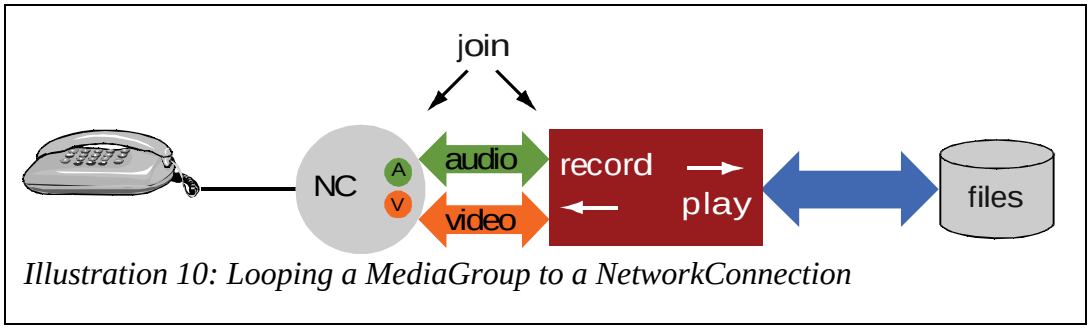


Illustration 10: Looping a MediaGroup to a NetworkConnection

You can also join two `NetworkConnections`, so that Mark and Fred can talk to each other, like this:

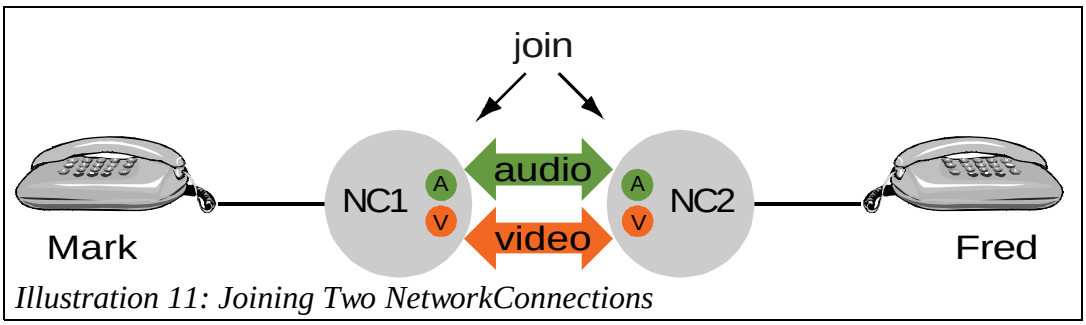


Illustration 11: Joining Two NetworkConnections

This method is usually referred to as a 'bridge' telephony service. Note that a `MediaGroup` is not needed, as there is no need to play or record anything.

If we compare a `MediaSession` with a room (for example, a conference room), then a `MediaGroup` is like a DVD player, and a `NetworkConnection` is like a phone on the desk. After the `NetworkConnection` has been set up and its description (SDP) negotiated, the phone's handset is “live”.

Joining the `NetworkConnection` to the `MediaGroup` is like putting the handset right next to the DVD speaker/screen. You can have multiple phones, and you can have multiple DVD players, or radio sets, VCR's or TV sets on your desk, or even text to speech (TTS) machines.

Ringing the phone, taking a call or placing a call, is outside the scope of JSR 309. JSR 309 currently only deals with the phone's screen and speaker.

Conferencing requires a big box, named a `MediaMixer`, with multiple slots. Each slot can fit either a handset, or the speaker and the screen of a VCR.

3.3 Join Options

In the previous examples, both audio and video streams were connected together bidirectionally, shown by the two arrows in the illustration.

You can also restrict the connection to a single direction, using an option to the `join` primitive:

In a telephony service, this is appropriate for an application that does not need to record messages. In this case the `MediaGroup` can be stripped out of the recorder resource. If a recorder is present,

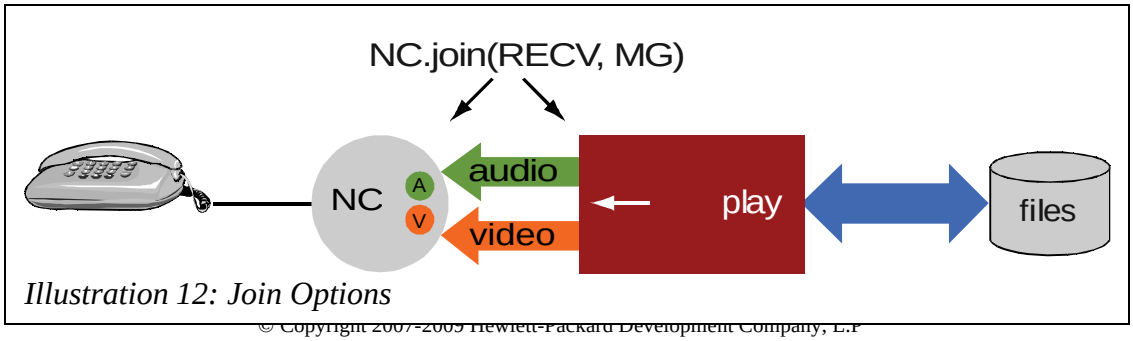


Illustration 12: Join Options

that is not considered as an error, but no recording is possible.

The direction is the first argument of the join method,. This means that:

```
NetworkConnection.join(RECV, MediaGroup);
```

reads “network connection ***receives from*** MediaGroup”.

It is equivalent to:

```
MediaGroup.join(SEND, NetworkConnection);
```

You can use either form, depending on how you visualize the objects.

You can restrict the connection to a single stream. To do this, take a handle on the stream (on the channel/port that carries this stream), using `getJoinableStream`, and then use this handle instead of the original object.

The example below combines direction and stream restrictions:

```
NC1.getJoinableStream(audio).join(DUPLEX, NC2.getJoinableStream(audio))
NC1.getJoinableStream(video).join(RECV, NC2.getJoinableStream(video));
```

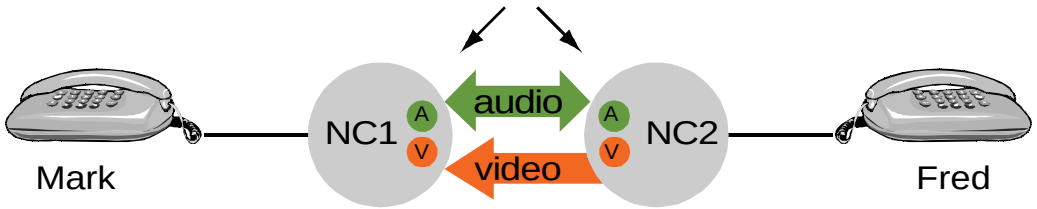


Illustration 13: Direction and stream restrictions

From a telephony service point of view, this is a special case of a 'bridge', where Mark and Fred can talk to each other, and Mark can see Fred. Fred cannot see Mark because Mark has subscribed to a 'privacy' option.

3.4 Multiple Joins

A media object can be joined to more than one media object.

Below is an example where a `NetworkConnection` is joined to two `MediaGroups`:

In this case, you can handle the audio and the video independently, each with its own dedicated

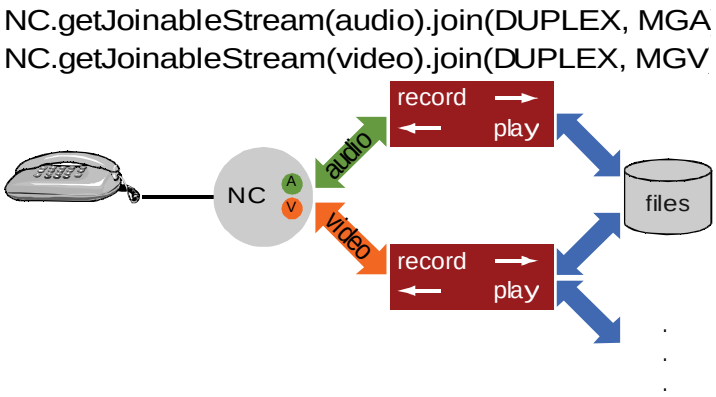


Illustration 14: Joining a NetworkConnection to Two Media Groups

`MediaGroup`. This method is referred to as 'split-stream media' and makes it possible, to use multiple media objects simultaneously. For example, you could use this to play a TTS prompt on the audio, at the same time as a synthesized avatar on the video (because a single `MediaGroup` cannot run two players simultaneously).

You can even join the **same object** multiple times:

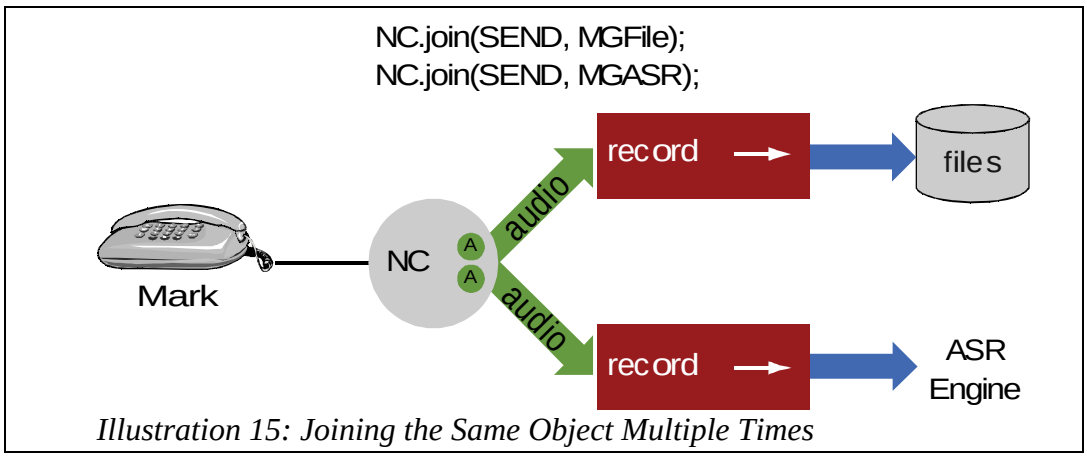


Illustration 15: Joining the Same Object Multiple Times

In this example, the media stream that goes out of NC is duplicated and sent to both `MediaGroups`.

You can use this in a telephony service, for example, to record Mark's message into a file, and simultaneously perform Automatic Speech Recognition on his voice.

JSR 309 can send or 'broadcast' a stream to as many recipients as desired. This applies to both video and audio.

Receiving from multiple sources is not possible, as it requires digital signal processing (DSP)

resources to perform 'additions' of the media samples, or 'mix' them. This is the role of the `MediaMixers`, and only those objects can take multiple inputs.

In summary, `NetworkConnections` and `MediaGroups` can '**talk to many**', but can only '**listen from one**'.

The following figure illustrates an example of this scenario:

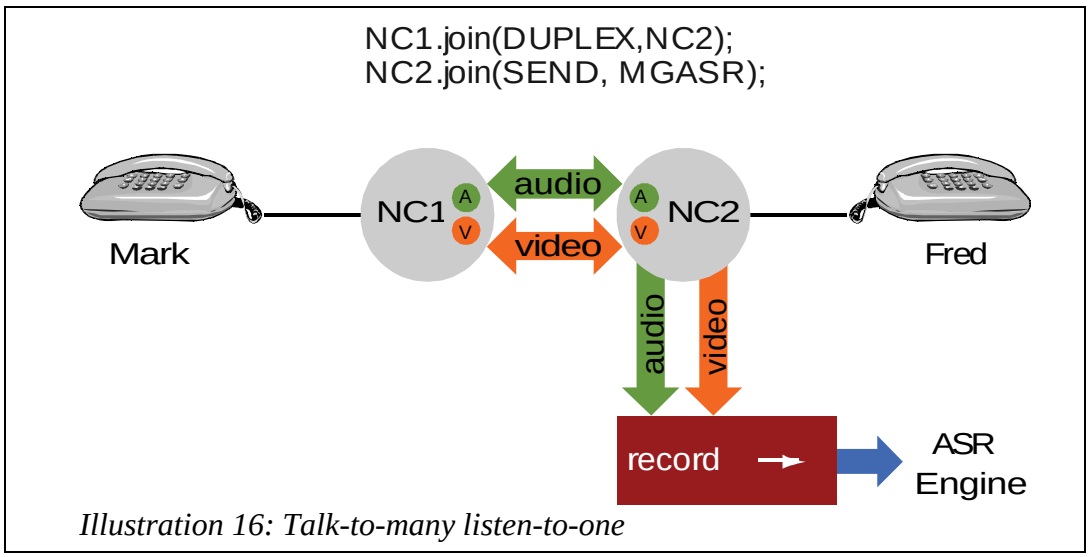


Illustration 16: Talk-to-many listen-to-one

After NC1 and NC2 have been joined to 'bridge' Mark and Fred, the application joins a `MediaGroup` to 'listen' to what Fred says, and sends it to an automatic speech recognition (ASR) engine. This does not require any media mixing, as NC2 just 'talks' to two objects.

In a telephony service, this could be used to cancel the bridging, for example when Fred says a keyword.

3.5 Join and Re-join Degradation

It is possible to attempt a sequence of join operations that request that a `NetworkConnection` 'listens' to more than one source. In this case, the previous join is automatically **degraded** to keep only the most recent listener.

For example, after bridging Mark and Fred, if the application joins bidirectionally MG1 to NC2, the result is shown below:

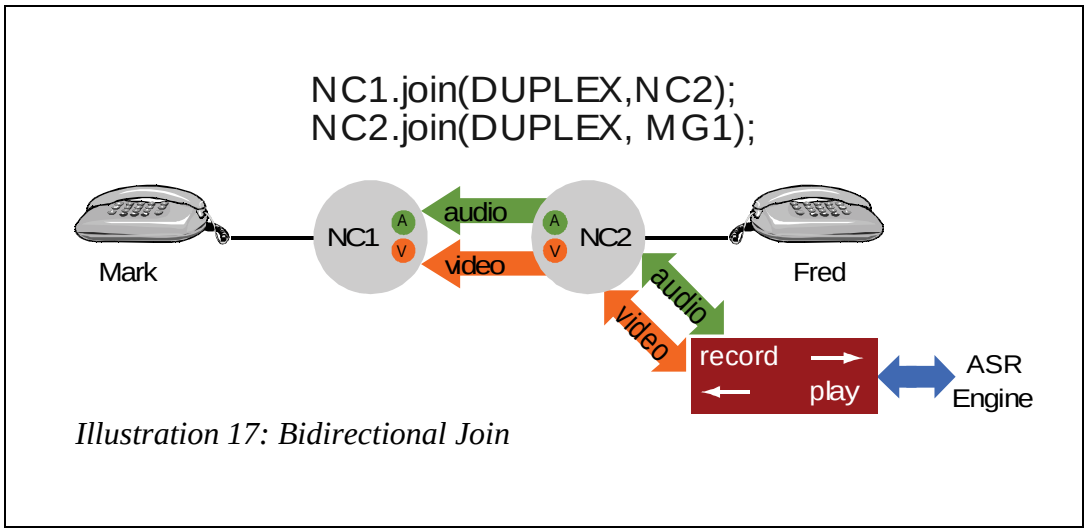


Illustration 17: Bidirectional Join

The media cannot flow anymore from NC1 to NC2, just as if NC1 had been joined to NC2 in RECV direction. From a telephony service point of view, Fred cannot hear Mark, but he can hear a prompt played by MG1.

You can 're-join' two objects that are already joined to change the connection properties. For example, in the situation shown in Illustration 17: Bidirectional Join, you can restore the full-duplex communication between Mark and Fred, as shown in Illustration 16: Talk-to-many listen-to-one, using `NC1.join(DUPLEX, NC2)`.

You can also change the connection properties by un-joining first, and then joining again with different options. In this case, signal processing is discontinued during the time interval between 'un-join' and 'join'. For this reason, only use un-join when all connections must be stopped.

Using this configuration you can record what Mark says and what Fred says in two different files, handled by two different `MediaGroups`. If you need to record what both Fred and Mark say in a single file, you need to use a `MediaMixer` (below). From a media point of view, a 'two-party conference' is the same, as from a telephony service point of view, it is a 'bridge' involving only two parties.

3.6 Simple Conference with a MediaMixer

You must use MediaMixers when you need to “listen to more than one source”.

The first telephony service application is a simple conference:

The next figure is obtained using:

```
NC1.join(DUPLEX, myMediaMixer);
NC2.join(DUPLEX, myMediaMixer);
NC3.join(DUPLEX, myMediaMixer);
```

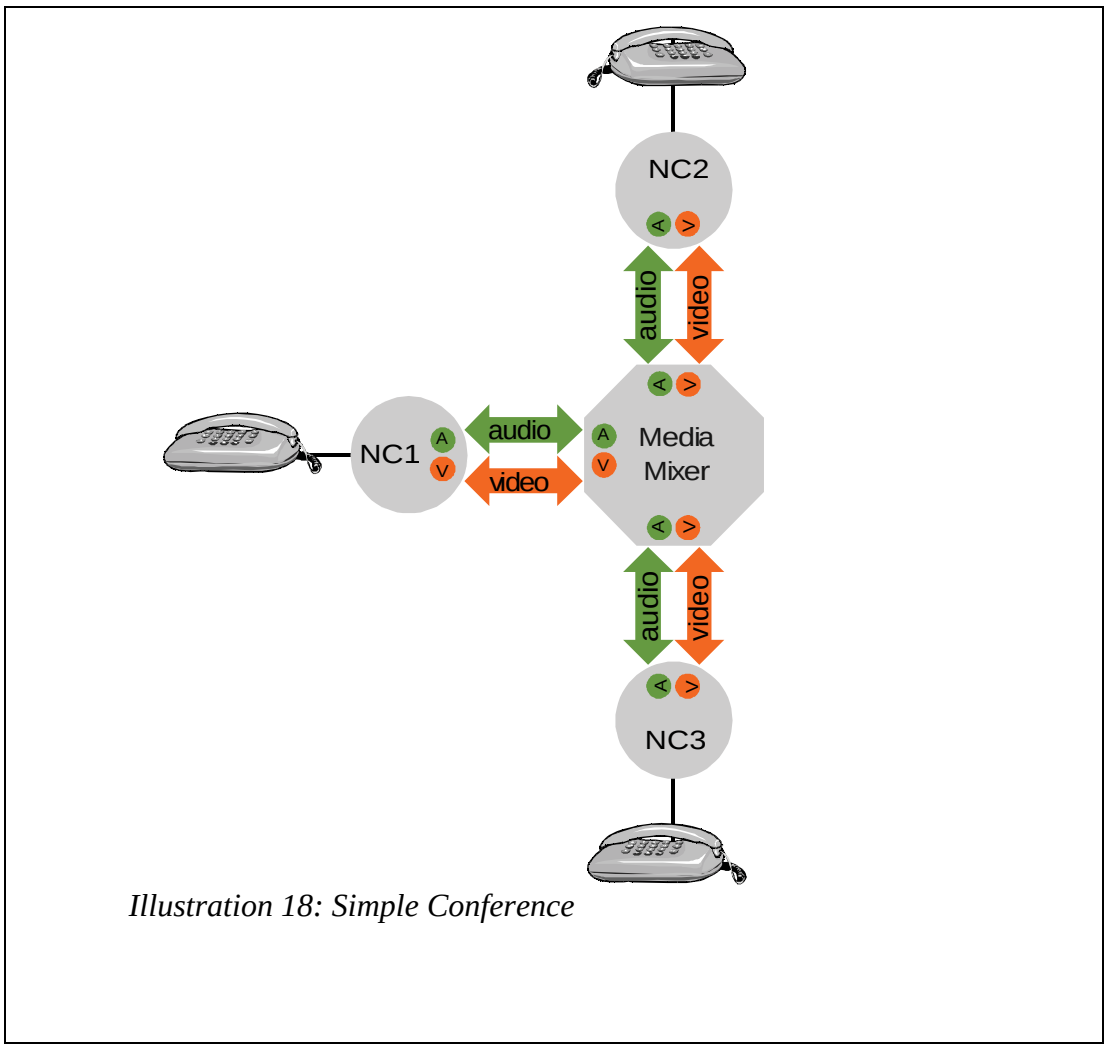


Illustration 18: Simple Conference

The result is that NC1, 2, and 3 now can talk to and see each other. Note that the various NC's can be associated with incoming calls, outgoing calls, scheduled outgoing calls, planned incoming calls, and so on. The `MediaMixer` can be created beforehand, or when all NC's are ready. A `MediaGroup` can have been joined previously to each NC, to play an introduction prompt. All these objects can belong to the same `MediaSession`, or to different sessions. This is a matter of telephony service, and it is entirely under the application's control and responsibility.

3.7 Listen-Only Participants

A conference participant can be put in “listen-only” mode (or “mute”) by restricting the join direction. This can be done at the first join of the NC, or afterwards, during the conference. The result is as follows:

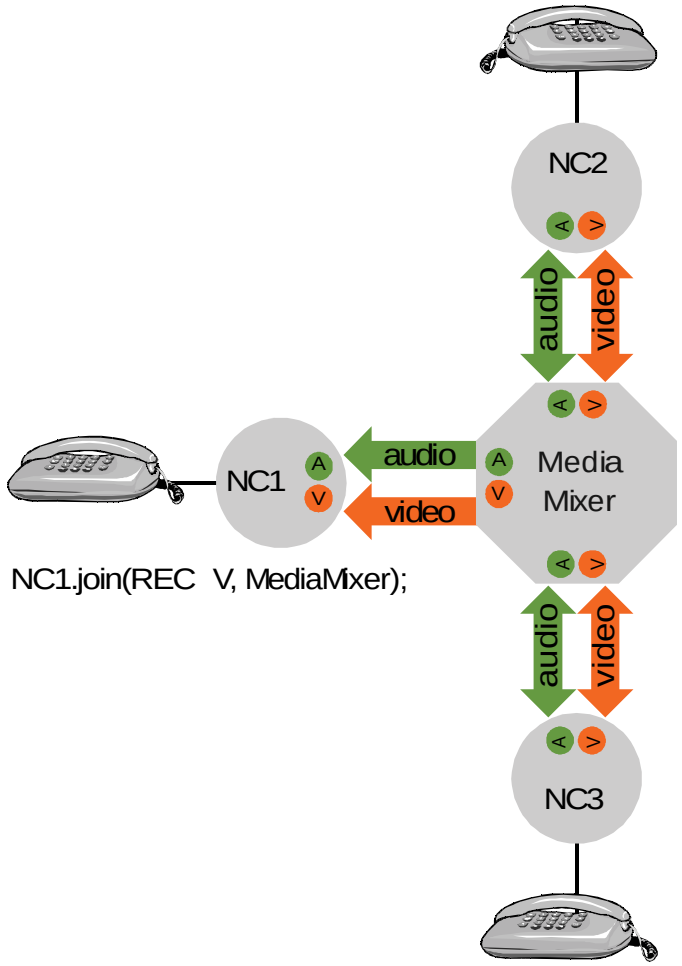


Illustration 19: Listen-only mode

You can mute only the video using:

```
NC1.getJoinableStream(video).join(RECV, myMediaMixer);
```

3.8 Recording a Conference, Playing Announcements into a Conference

To record a conference, or play announcements during a conference, you need to join a `MediaGroup` to the conference. You can use any available `MediaGroup`, or create a new one, with the set of resources that you need (TTS, etc).

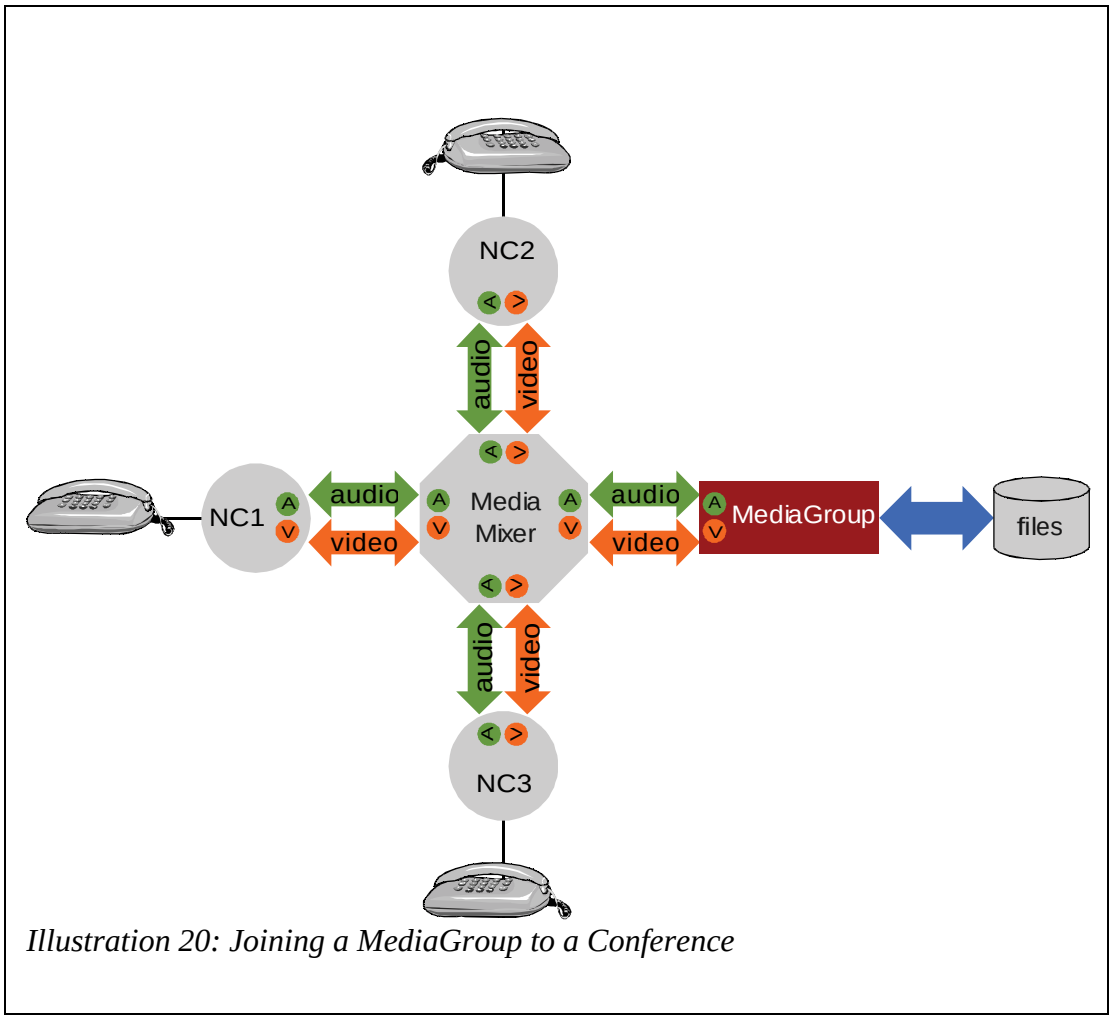


Illustration 20: Joining a MediaGroup to a Conference

The `MediaMixer` treats all the objects that are joined to it in the same way. Each of them receives the sum of all other participants (excluding itself). This also applies to the `MediaGroup` shown above. As a result, this `MediaGroup` records everything said in the conference, except the prompts that it plays itself. If the application needs to record these as well, it can join a second `MediaGroup` (one for plays, one for records).

3.9 Advanced Conference Features

This includes side-bar conferences, whispering, coaching, background music, and more. All of those can be performed by a combination of `NetworkConnections`, `MediaMixers` and

`MediaGroups`, using the appropriate join options. Some of them may require multiple `MediaMixers` joined together. In some use cases, several `MediaMixers` are required, even though the telephony service involves only one party.

3.9.1 MixerAdapters

Advanced conference applications require some media processing specific to each “conference leg”. For example, setting an input or output gain for the leg.

To do so using JSR 309, you need a media object that contains the corresponding DSP resources: a `MixerAdapter`. A `MixerAdapter` is a dependent object of a `MediaMixer`:

```
myAdapter = myMediaMixer.createAdapter(myConfig);
```

You can join to the `MixerAdapter`, like to any other `Joinable`:

```
myAdapter.join(DUPLEX, NC4);
```

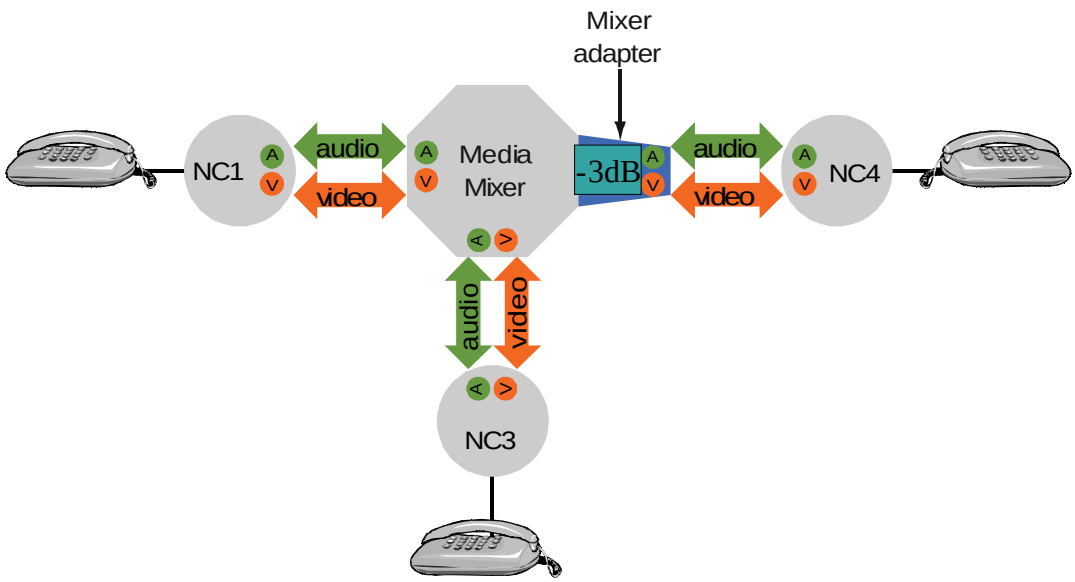


Illustration 21: Joining a MixerAdapter

A `MixerAdapter` is also capable of joining two `MediaMixers` (with or without extra media processing):

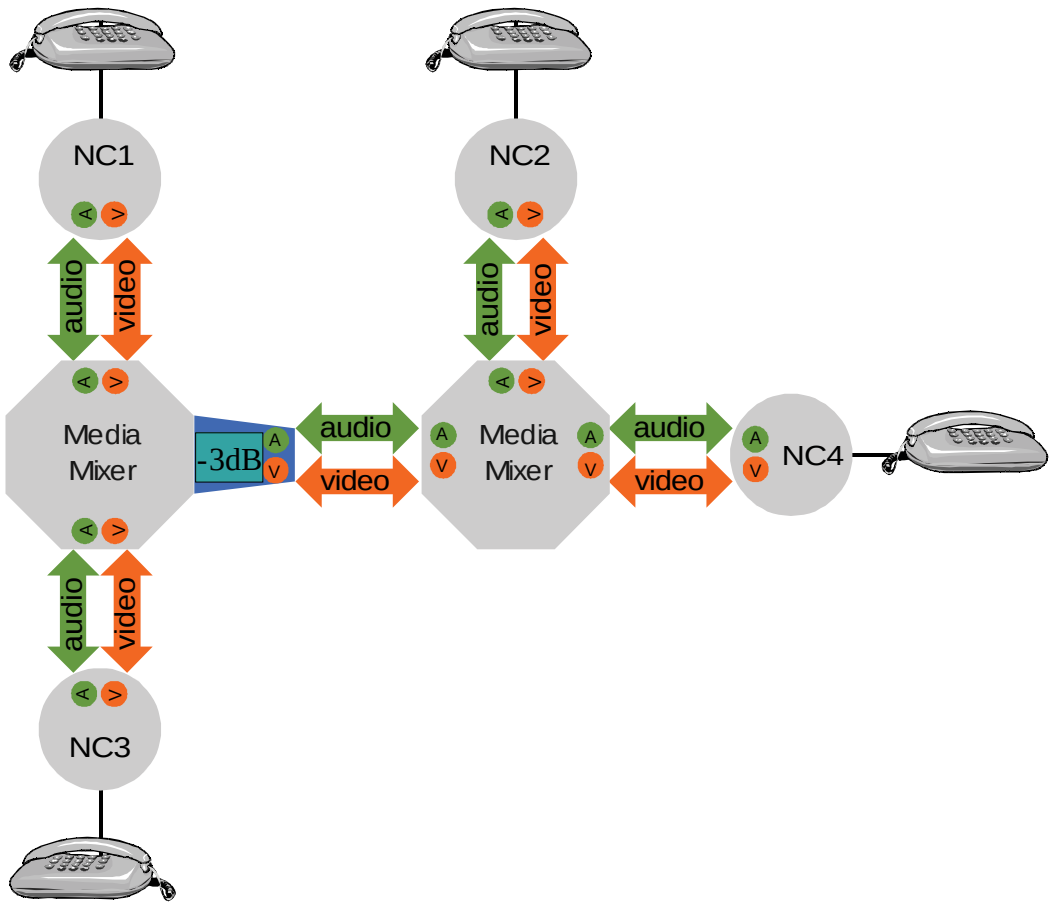


Illustration 22: Joining two MediaMixers

3.9.2 Example Applications

3.9.2.1 Recording a Conversation, Including Video

A Customer is bridged to an Agent in a call-center, with video. The application wants to record the conversation, in a single 3gp file. To keep it simple, only the customer's video is included in the file, however the audio is the combination of both inputs:

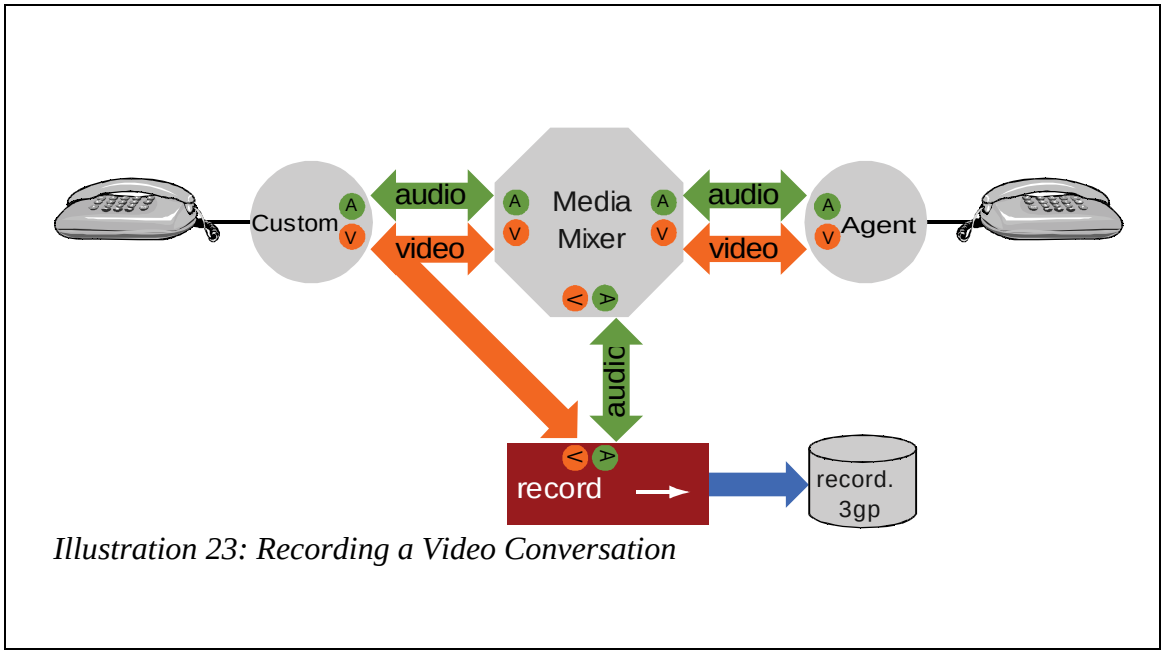


Illustration 23: Recording a Video Conversation

The code to do this is the following:

```
NetworkConnection customerNC, agentNC; // obtained previously
MediaMixer myMixer = createMediaMixer(MediaMixer.AUDIO_VIDEO);
MediaGroup myMG = createMediaGroup(MediaGroup.PLAYER_RECORDER_SIGNALDETECTOR);

customerNC.join(DUPLEX, myMixer);
AgentNC.join(DUPLEX, myMixer);
myMG.getJoinableStream(StreamType.audio).join(DUPLEX, myMixer);
myMG.getJoinableStream(StreamType.video).join(RECV, customerNC);
```

3.9.2.2 Whispering/Coaching

In this application, a Customer is calling a call-center Agent. Since the Agent is a beginner, a Coach is advising him/her, by whispering some indications. The Customer is unaware of the Coach's presence. The Coach can hear the whole conversation.

This can be achieved using a Mixer, implementing the private conversation between the Agent and the Coach. The Customer receives audio from the Agent directly:

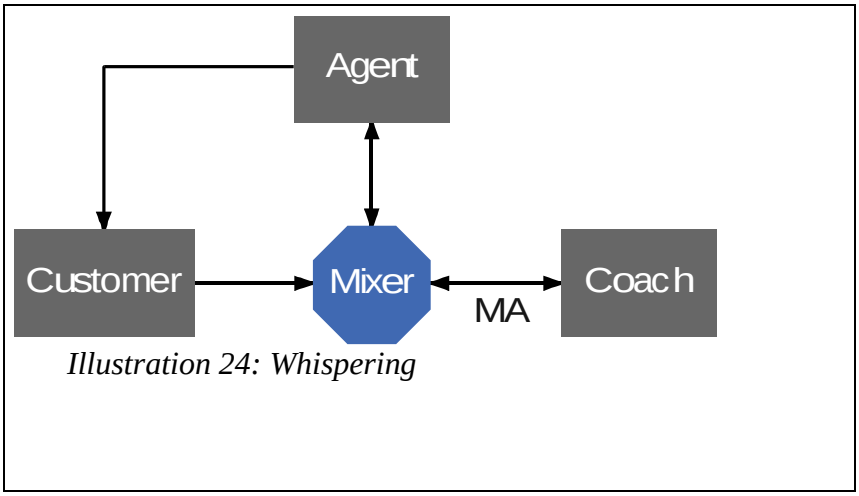


Illustration 24: Whispering

Sample code is below. Each "join" represents an arrow on the diagram:

```
MediaMixer privateM = createMediaMixer(MediaMixer.AUDIO_VIDEO);
NetworkConnection agentNC, customerNC, coachNC; // obtained previously

agentNC.join(   SEND, customerNC);
agentNC.join( DUPLEX, privateM);
customerNC.join(SEND, privateM);
coachNC.join( DUPLEX, privateM);
```

3.9.2.2.1 Video-enabling:

The above diagram would also work for video. The participants can see those they have permission to see. And since the code sample joins the objects globally (and not per-stream), the code is the same as well.

The default video layout of the mixer should be acceptable for this case. However it would be better to use `privateM` with a horizontal-two-tile layout. A tiling (or mosaic) layout needs many more CPU cycles. This can be left out as an option for high quality service.
Note that the quality perceived by the Customer will be optimal in any case, since he or she receives a single stream (audio+video), directly from the Agent, and no media processing is required.

3.9.2.2.2 Refinements

You might want that the Agent hears the Coach with a lower volume, so that he/she can concentrate on what the Customer says. This requires a `MixerAdapter` (named "`lowVolumeMA`") at point MA in the diagram. The three lines below replace the line that joins `coachNC` to `privateM`:

```
MixerAdapter lowVolumeMA =
privateM.createAdapter(MixerAdapter.DTMFCLAMP_VOLUME);
coachNC.join(DUPLEX, lowVolumeMA);
```

The volume can be set to an initial value, by customizing the `MediaConfig` of the `MixerAdapter` and/or it can be adjusted at any time using:

```
lowVolumeMA.triggerAction(VolumeConstants.VOLUME_UP); // or VOLUME_DOWN
```

3.9.2.2.3 More participants – Generalization to Sidebar Conferences

The example below assumes two Coaches, who can listen to the conversation, talk to the Agent, and talk to each other as well. If we keep the same approach, the diagram becomes:

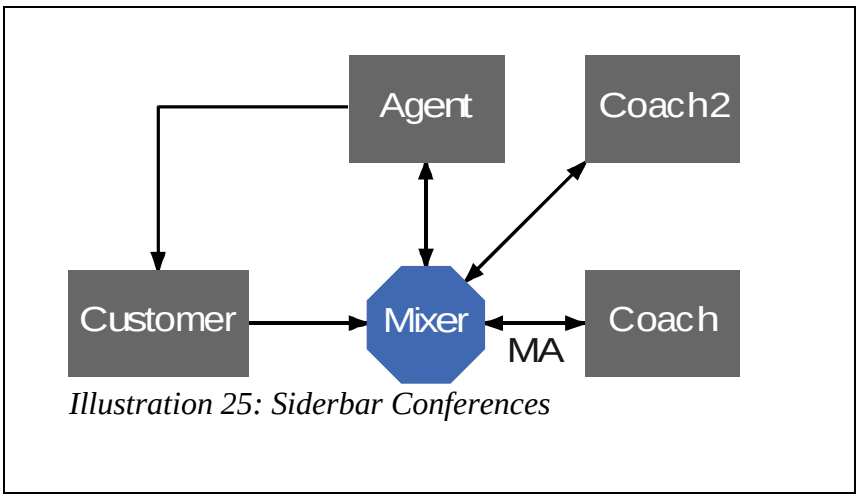


Illustration 25: Siderbar Conferences

The code for this is straightforward.

You might want multiple customers, and two distinct "conferences": a main "public" conference, to which the Customer(s) and the Agent(s) are connected, and a "sidebar" conference, for the Coaches(s). The sidebar conference can listen to the main conference (but not vice-versa). The Agents talk to the main conference, but listen to the sidebar conference:

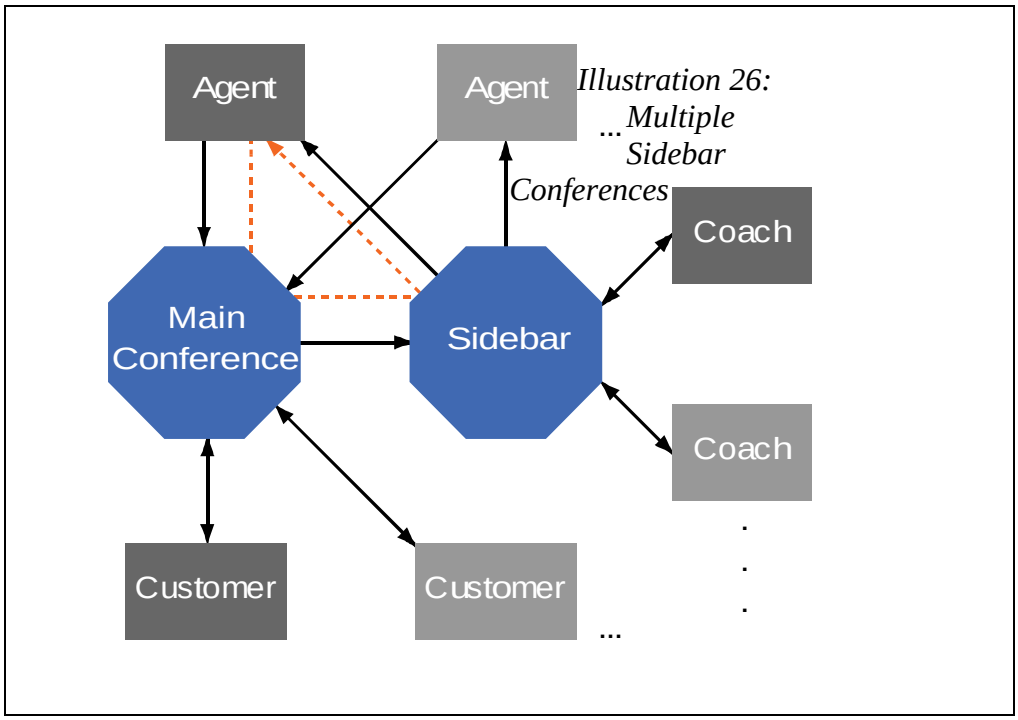


Illustration 26: Multiple Sidebar Conferences

This requires only two Mixers, and scales well with the number of participants.

However it has an unpleasant side-effect. The media samples output by the Agent follow the dotted red line, and come back to the same participant, causing a disturbing echo.

To avoid this echo (assuming you do not want to rely on echo-cancellation), the Agent needs to listen to the same "port" of the main conference as he is talking to (using an empty `MixerAdapter` to represent the port). To do this, an adder is needed, for combining with sidebar input:

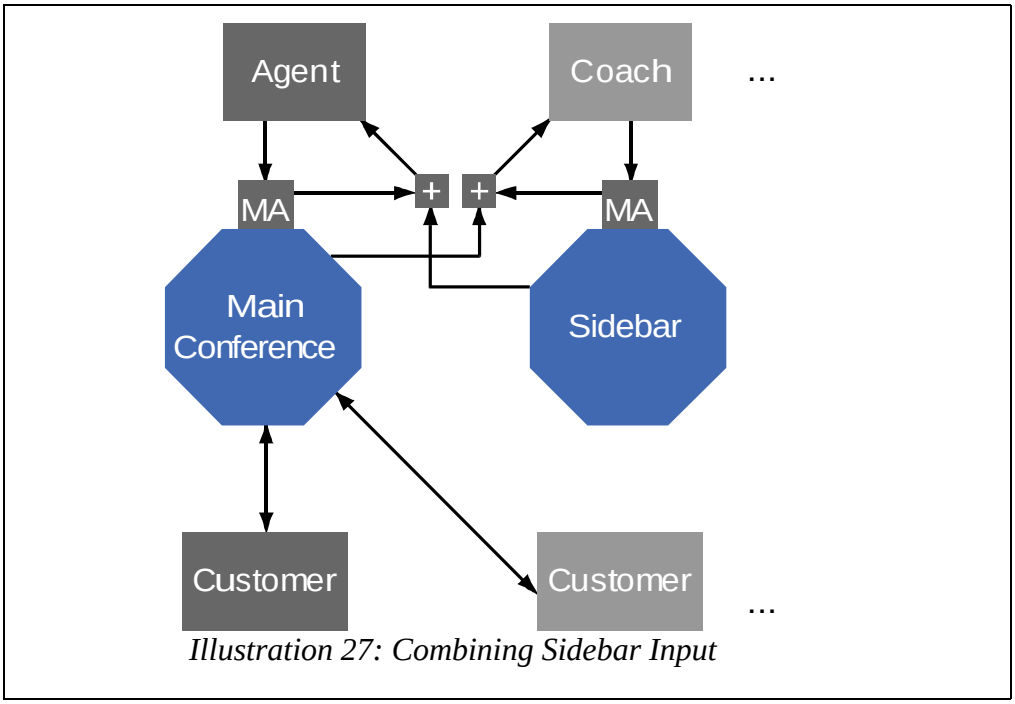


Illustration 27: Combining Sidebar Input

Note: an `adder` is a mixer that has multiple inputs but a single output, represented here with a "+" sign for clarity. In JSR 309 this is implemented using a plain `MediaMixer` (it is up to the implementation to optimize for this particular use-case).

Each Agent and Coach requires a `MixerAdapter` and an `adder`, but the number of objects and operations now scale linearly with the number of participants. Note that there is no feed from the main to sidebar conference anymore (this is what caused the echo).

A simpler application would not include the role of Agent. All participants are either Customers or Coaches. The Coaches have their own private chat session, and can hear the main conference in the background:

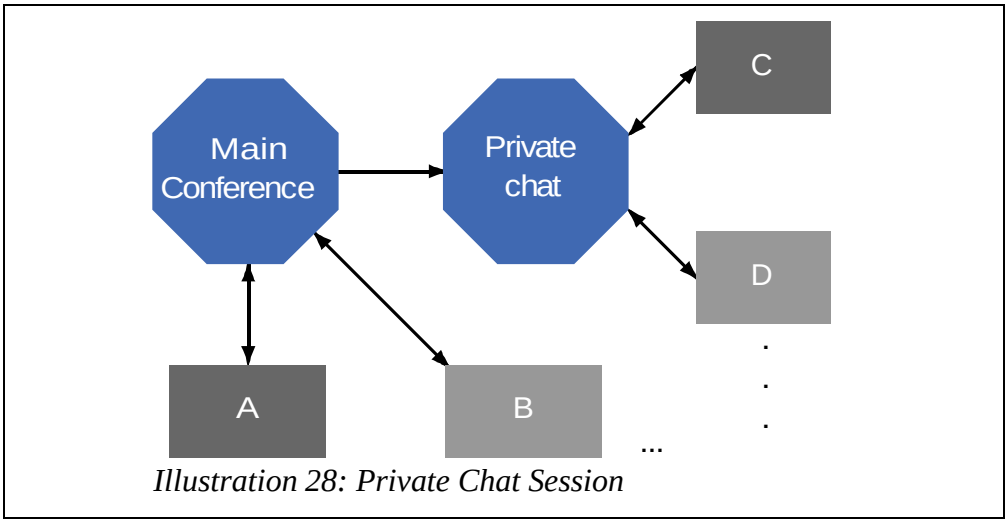


Illustration 28: Private Chat Session

It is possible to add other features such a decreased volume between the main and the sidebar conference, muting the main in the sidebar, changing the role of a participant, cascaded sidebars, and so on.

3.9.2.3 Video-Enabled Sidebar Conferences

3.9.2.3.1 Simple Mixers (Video Switching)

A simple mixer does not combine any special video streams, it just selects one input stream, and sends it to everybody. Usually the selected input is the participant that speaks loudest.

In the scheme shown in Illustration 25: Siderbar Conferences, each person sees the person who talks loudest. This is computed individually for each participant. Alternatively, a DTMF key could let each participant select the person to see. The DTMF key would result in a `setParameter()` call to the `MediaMixer`, to program the switching.

Using the scheme shown in Illustration 26: Multiple Sidebar Conferences, each Customer sees the Agent who talks loudest, and this is what we want anyway. The Agents and Coaches will see either the loudest Coach, or the loudest among the Agents/Customers. This includes a cascaded selection (when the loudest contributor of the Sidebar is the `MainConference`, where a selection of the loudest talker is also taking place), and if a volume control is applied between the main and sidebar, it will affect the selection. The outcome is non-deterministic.

Using the scheme shown in Illustration 27: Combining Sidebar Input, we still have a cascaded selection process.

In the case of a simple private chat, as in Illustration 28: Private Chat Session, there are two options:

1. The private chatters have little interest in the main conference, and the join between the main and the sidebar conference is “audio-only”. Each group has its own video selection.
2. The private chatters are interested in the video of the main conference, and want an audio-only chat session. In that case, they would be joined audio-only to the private chat session, and video-only (see-only) to the main conference.

A combination of both modes (per-participant choice) is another possibility.

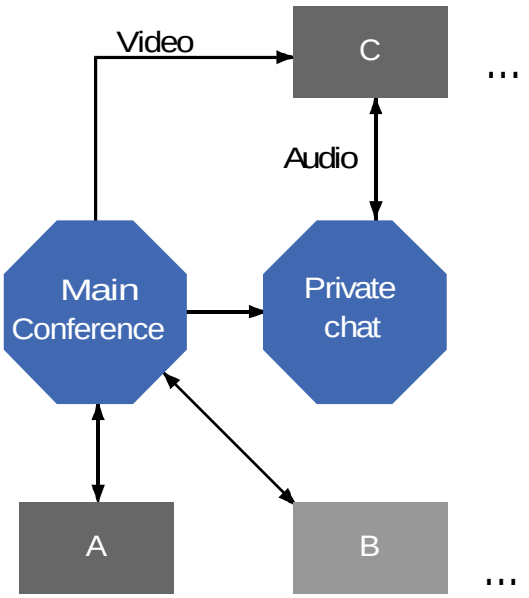


Illustration 29: Private Chat (audio only)

3.9.2.3.2 Mixers with Video Layout Control

Mixers with video layout control can make a tiled/mosaic image, with multiple participants, decoration, etc. Their use in the context of sidebar conferences is not explored yet, and very application-specific. A very flexible layout description enables any combination.

3.10 Complex User Interactions: VoiceXML Dialogs

A `MediaGroup` provides a pure Java interface, to provide a dialog with a telephone user. Some applications may prefer to use VoiceXML for that purpose. In that case, the user interactions are described in a vxml document, and JSR 309 provides the support of a `VxmlDialog` object. Its methods are `prepare`("http://appserver/welcome.vxml"), `start()`, and `exit()`. The application registers a listener on the `VxmlDialog` object, to get notified of termination, and possible mid-dialog events.
A `VxmlDialog` is a Joinable object, and can be used in place of a `MediaGroup`, anywhere in this document. For example:

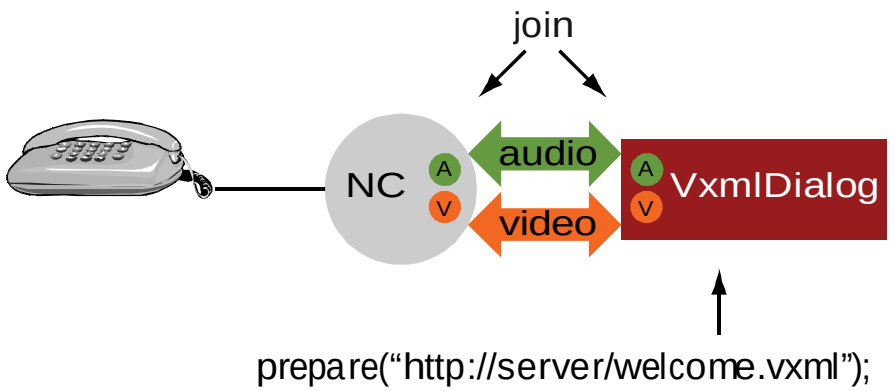


Illustration 30: VxmlDialog in place of MediaGroup

4 MediaGroup Features

The `MediaGroup` object is a multi-purpose player/recorder/dtmf-detector/generator. It can be viewed as a combination of 4 resources:

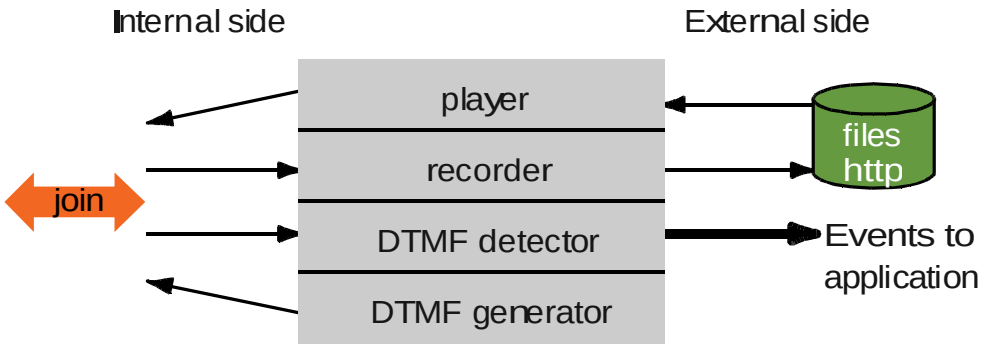


Illustration 31: MediaGroup Overview

The `MediaGroup` object acts as an itermediary between external media (mostly files) and the other media objects defined by JSR 309 (`NetworkConnections`, `Mixers`).
An application needs a `MediaGroup` for simple interactions with a user: playing a file, detecting DTMFs, etc. For more complex interactions, it can use a `JoinableDialog`, like the vxml dialog object described in section 3.10 Complex User Interactions: VoiceXML Dialogs.
In all cases, the `MediaGroup`/`JoinableDialog` must be joined to `NetworkConnection`(s) and/or `MediaMixer`(s), as described in section 3 Media composition: Basic IVR, bridging and mixing API.
A `MediaGroup` is a basic dialog, with capacities such as `play()`, `record()`, and so on.

4.1 MediaGroup Interfaces

4.1.1 Player

The Player resource can be accessed using `MediaGroup.getPlayer();`

The main method of this interface is

```
void play(URI streamID, RTC[] rtcs, Parameters optargs);
```

For example:

```
play(URI.create("/mediafiles/myfile.3gp"), RTC.NO_RTC, Parameters.NO_PARAMETER);
```

- The RTC array defines the Run Time Controls associated with the operation. See section 4.4 RunTimeControls (RTCs) for more about RTCs.
- *optargs* is a map of `{Parameter, Object}` , to provide tunings specific to this transaction.
- This method starts the play operation, and returns immediately. When the transaction is

completed (e.g., end of file), the application is notified through a listener.

Note: for those used to xml-based interfaces, that would translate as:

```
<play param=”value” param1=”value1” >
  <uri>/mediafiles/myfile.3gp</uri>
</play>
```

4.1.2 Recorder/SignalDetector/Generator

They have similar interfaces, with a few main arguments, and a *Parameters* map for options:

```
void record(URI streamID, RTC[] rtcs, Parameters optargs);

void receiveSignals(int numSignals, Parameter[] patternLabels, RTC[] rtc,
Parameters optargs);

void emitSignals(String signals, RTC[] rtcs, Parameters optargs);
```

4.2 Parameters

In the *Parameters* map above, each parameter is identified by a `Parameter` class. `Parameter` are non-mutable tokens, similar to Java `enum` members. In the `play` case, you can use any of the `Parameter` in `javax.media.mscontrol.mediagroup.Player.java`. For example, `MAX_DURATION`.

The value associated to a `Parameter` is a typed object, whose type depends on the parameter itself. In the case of `MAX_DURATION`, the value must be an Integer, in milliseconds. Refer to the javadoc for the list of `Parameter`'s and value types.

This sort of interface has several advantages:

- Powerful, because the values are typed objects; most parameters are Boolean, Integer, or list-valued; some are full-blown typed objects, like an SRGS grammar for `SignalDetector.PATTERN`.
- Extensible at wish, by adding new `Parameter`'s, and defining their value. As an example, `javax.media.mscontrol.mediagroup.http.HttpFilePlayerConstants` adds the parameters specific to http, like `FETCH_TIMEOUT` and `FILE_MAX_AGE`. Some extensions can be MS-vendor specific, and they will not break mainstream applications.

However the disadvantage is that type checking is performed at run time, not at compile time.

4.2.1 Code samples:

$ Setting the time boundary to 10s for a play transaction:

```
Parameters playParams = myMediaSessionFactory.createParameters();
playParams.put(Player.MAX_DURATION, new Integer(10000));
myMediaGroup.play(myURI, RTC.NO_RTC,  playParams);
```

$ Record a file in oki encoding:

```
Parameters recordParams = myMediaSessionFactory.createParameters();
recordParams.put(Recorder.AUDIO_CODER, CoderConstants.ADPCM_32kOKI);
myMediaGroup.record(myURI, RTC.NO_RTC,  recordParams);
```

As shown above, some parameters are list-valued (equivalent to a `restriction base=NMTOKEN` in xml parlance). Those items are described by a Value instance, like `ADPCM_32kOKI` above.

4.2.2 Permanent Settings: `setParameters` Method

You can also set some parameters **permanently** for a given `MediaGroup`, using:
`MediaGroup.setParameters(Parameters params);`

For example, if you want to limit recordings to 2 minutes, and playbacks to 5 minutes:

```
Parameters limits = myMediaSessionFactory.createParameters();
limits.put(Recorder.MAX_DURATION, new Integer(120000));
limits.put(Player.MAX_DURATION, new Integer(300000));
myMediaGroup.setParameters(limits);
```

You do not need to repeat the max duration for each `play()` or `record()` call anymore.

4.3 Using the SignalDetector

The SignalDetector has a very versatile interface, allowing many combinations. Below are the most common, by complexity order.

In each case, the application must register a listener to be notified of the SignalDetectorEvents.

4.3.1 DTMF subscription

Some applications want to be notified of every DTMF as soon as it is detected, and implement any pattern matching or processing of their own.

This is programmed by setting ENABLED_EVENTS to SIGNAL_DETECTED. On top of this, disabling buffering might help performance and avoid buffer overflow: set BUFFERING to false.

4.3.2 Patterns definition

The application defines patterns as Parameters, as explained in the javadoc. The Parameter's names are PATTERN[0], PATTERN[1], ...etc. For example:

```
Parameters params = mySigDet.createParameters();
params.put(SignalDetector.PATTERN[0], “*”);
myMediaGroup.setParameters(params);
```

Most applications use a limited number of patterns, that can be all defined when the MediaGroup is created. Later, at each phase of the user interaction, the application may enable some or all of them.

4.3.3 Simple Pattern matching

The application can register for the event associated to the matching of any pattern, for example PATTERN[0]:

```
params.put(SignalDetector.ENABLED_EVENTS, PATTERN_MATCHED[0]);
myMediaGroup.setParameters(params);
```

(this can be combined with the previous setting).

Multiple patterns may be defined and activated simultaneouly.

The application will receive one `PATTERN_MATCHED` event for each match. The event will indicate the pattern index, and the DTMFs that were collected so far.

Note that some implementations may not support this feature, as it is optional.

4.3.4 receiveSignals

The method `receiveSignals` is a compound operation, that can complete for a number of reasons, like the number of DTMFs received, various timeouts, patterns matching.
All the settings are defined as arguments of the method. (except the definition of patterns, that must be done previously as explained before)
Please refer to the javadoc for the exact meaning of each argument.

When the operation is complete, the application is notified with a single event `RECEIVE_SIGNALS_COMPLETED`, and the collected DTMFs are available in the event.

`receiveSignals` is adequate to run a moderately complex operation, like collecting a PIN number, or navigating through menus.

4.3.5 Prompt and Collect, Prompt and Record

Many IVR applications need to play a prompt, and then collect some DTMFs or Speech input. While the application can use `play` followed by `receiveSignals`, it is convenient to use a single operation to do so.

The parameter `SignalDetector.`**`PROMPT`** allows a prompt to be played as a preliminary to the DTMF collection performed by `receiveSignals(...)`, thus providing a prompt-and-collect primitive. See the javadoc of `PROMPT` for more information.

Similarly, `Recorder.PROMPT` provides a prompt-and-record feature.

4.4 RunTimeControls (RTCs)

An RTC class is simply the association of a `Trigger`, or condition (e.g., DTMF detection) with an `Action` (e.g., stop the player). When the condition occurs, the `MediaGroup` performs the associated action without requiring intervention of the application. Most current uses include barge-in (DTMF==>stop-player), raise the volume when `6` is entered, jump forward when `9` is entered ...

Code example:

```
RTC bargeInRTC = new RTC(SignalDetector.DETECTION_OF_ONE_SIGNAL, Player.STOP);
myMediaGroup.play(myURI, new RTC[]{bargeInRTC}, null);
```

All four MediaGroup transactions accept an (optional) array of RTCs. In addition, the application can "manually" trigger an action, using `MediaGroup.triggerAction`. For example, to pause the player:

```
myMediaGroup.triggerAction( Player.PAUSE);
```

Like the `Parameter`'s, the list of `Trigger`'s and `Action`'s can be extended by adding new items.

4.4.1 RTC versus MediaEvents

As shown in Illustration 32: RTC compared with MediaEvents, an RTC, when it fires, is executed completely internally to the `MediaGroup`, without invoking the application. This allows fast reaction, regardless of the protocol used to talk to the media server.

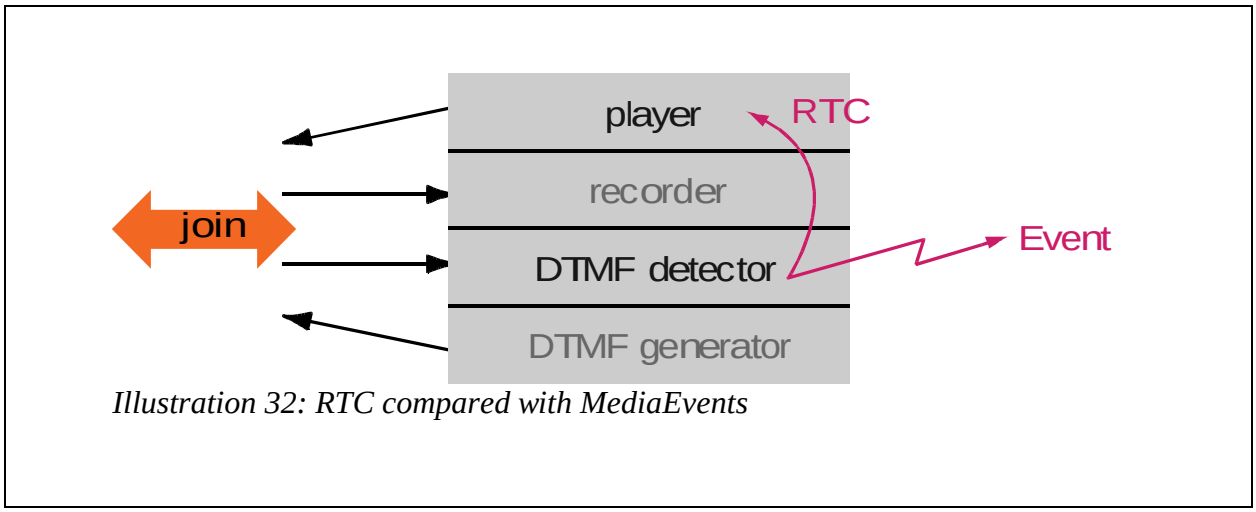


Illustration 32: RTC compared with MediaEvents

The action associated with an RTC must be defined by an `Action` instance, while a MediaEvent can be associated to an application-provided piece of Java code, of any complexity.

RTCs and MediaEvents are defined independently (see the next paragraph for events). The same media event can cause both an RTC and a MediaEvent, or just one of them:

- In the barge-in example, we want the player to stop immediately (an RTC) and also that the application is notified (a MediaEvent) to collect the digits.
- In the "press 6 to raise the volume" example, an RTC would associate the condition "DTMF 6 received" to the action `Player.VOLUME_UP`. In that case, a MediaEvent to the application is not particularly useful.

4.5 Events and listeners

Each of the transactional resources (Player, Recorder, SignalGenerator, SignalDetector) can accept an application-coded listener. The listener has a single method, `onEvent`, which delivers a structured event to the application. All listeners are instances of the template class

`MediaEventListener<T extends MediaEvent>.`

The events form a hierarchy, with the base providing the basic fields, and the extensions providing case-specific details:

MediaEvent (provides an eventID)
- **ResourceEvent** (provides an error code, a qualifier, and a RTCTrigger)
  - **PlayerEvent** (provides the offset into the file)
  - **RecorderEvent** (provides the duration of the recording)
  - **SignalDetectorEvent** (provides the DTMFs received)
  - **SignalGeneratorEvent**

Code snippet for detecting DTMF's:

```
MediaEventListener<SignalDetectorEvent> myListener =
                new MediaEventListener<SignalDetectorEvent>() {
      void onEvent(SignalDetectorEvent event) {
            if (event.isSuccessful()) {
                  System.out.println("Received "+event.getSignalString());
            }
      }
}
myMediaGroup.getSignalDetector.addListener(myListener);
```

4.6 Optional features

4.6.1 Resource Configuration

A `MediaGroup` is a collection of resources; Some resources source the data (the Player type), some sink the data (the Recorder type), and some are just mid-stream data filters/transformers: the latter category include DTMF detectors, speech detectors, volume controls, and so on.

The list and configuration of Resources that compose a given `MediaGroup` is defined by a `MediaConfig` class. The `MediaConfig` is a constructor argument of a `MediaGroup`.
A single instance of a `MediaConfig` can generate as many `MediaGroups` as required. It is a read-only, descriptive object, like a construction template. Typically an application uses a single, static instance of `MediaConfig`.

For general use, the API provides ready-made/predefined configurations, like `MediaGroup.PLAYER_RECORDER_SIGNALDETECTOR`, (suitable for an IVR-type application).

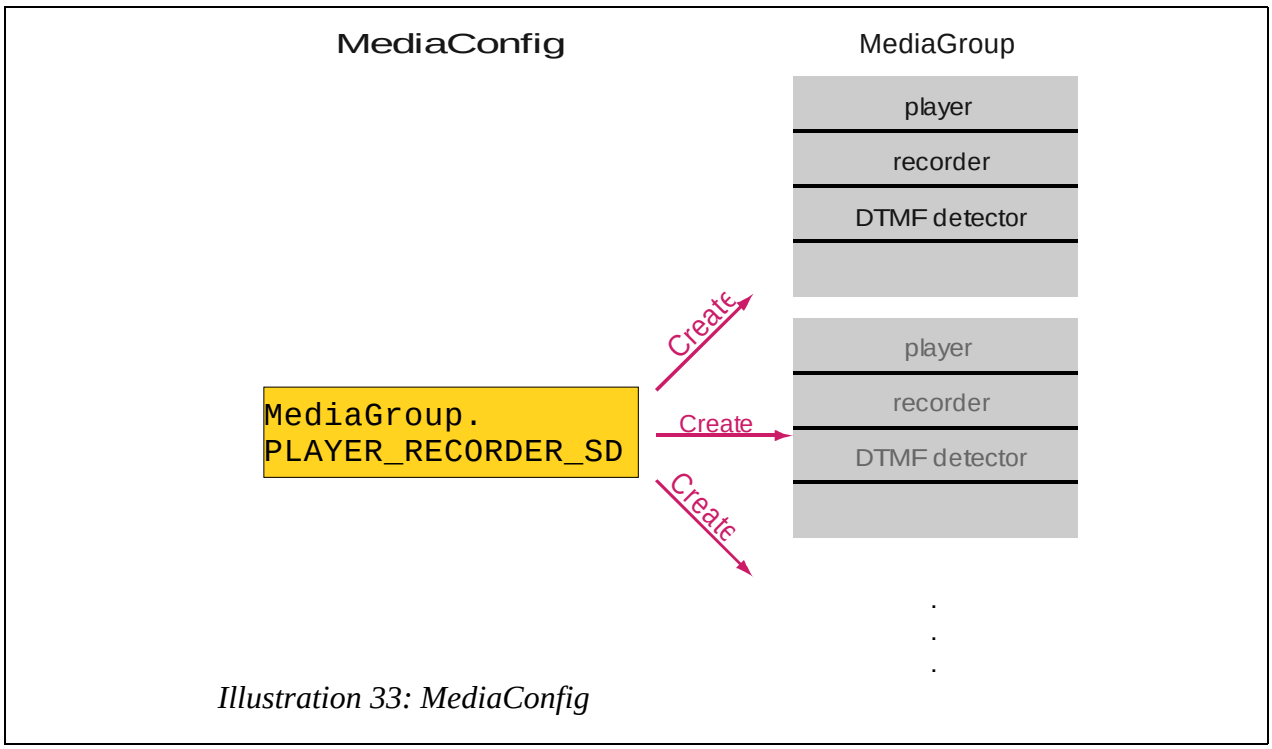


Illustration 33: MediaConfig

For more specific usages, the application can provide its own specific configuration (adding or removing items). In this case, the application must build a new `MediaConfig`, providing a definition for it. The definition can be a property file, or an XML structure as in the diagram below:

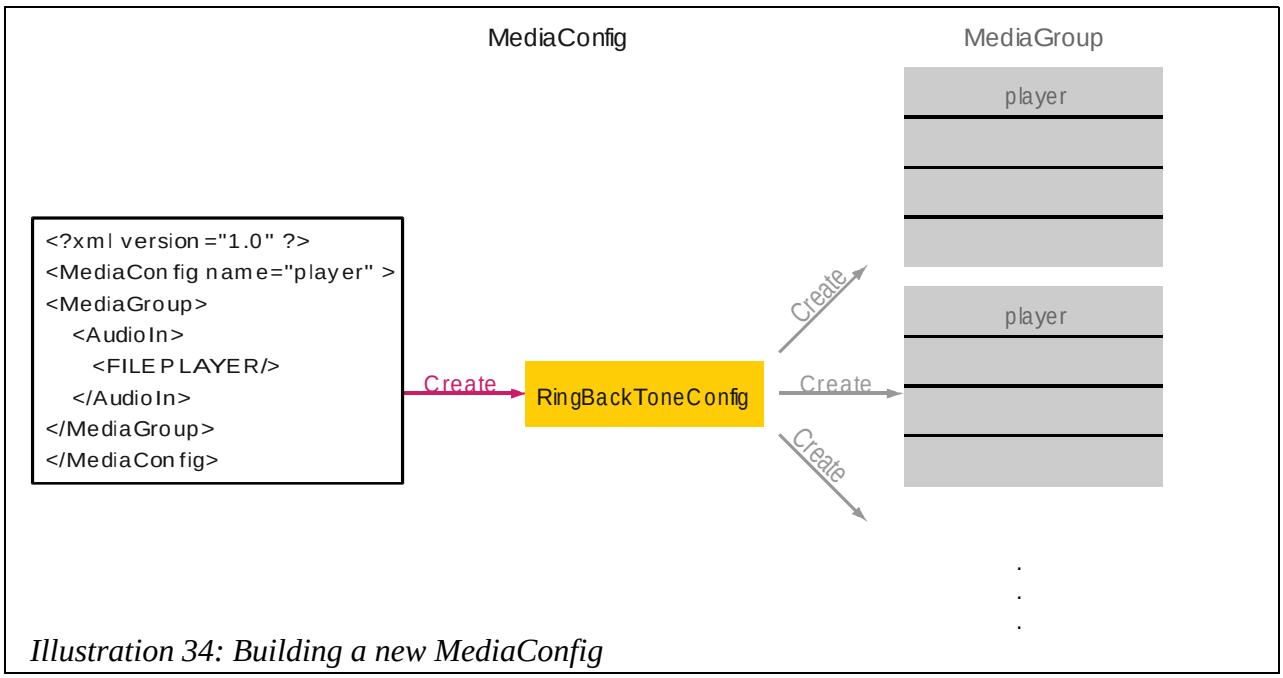


Illustration 34: Building a new MediaConfig

This feature is optional, and the format of the definition is vendor-specific. The method `MediaSessionFactory.getMediaConfig` is required to do that.

4.6.2 Setting parameters in the MediaConfig

The main use of the `MediaConfig` is to provide the name and arrangement of the resources contained in the resource group.
However the `MediaConfig` can also be preset with specific values for the parameters of the contained resources. This is yet another way to customize the `MediaGroup` parameters. You can build a customized `MediaConfig` from an existing `MediaConfig`. You first need to build a Parameters class containing the settings, as in paragraph B.2.2:

```
Parameters bounds = myMediaSessionFactory.createParameters();
bounds.put(Recorder.MAX_DURATION, new Integer(120000));
bounds.put(Player.MAX_DURATION, new Integer(300000));
```

This Parameters class is then a suitable argument to the *customize* method:

```
MediaConfig boundMediaConfig = ivrMediaConfig.customize(bounds);
```

`boundMediaConfig` will have the same list and arrangement of resources as `ivrMediaConfig`; But when you create a `MediaGroup` with `boundMediaConfig`, the `MAX_DURATION` values will be set from the start.

The `customize` method can be applied to any `MediaConfig`, either predefined or specific.

4.6.3 Extensibility

A new Resource can be added by building a `MediaConfig` to contain it, and a set of `Parameter`'s, `Action`'s, `Trigger`'s to control it: for example, a `VoiceModulator` could be provided by a `MediaConfig` of MediaGroup that adds it right after the player, and a `VoiceModulatorConstants` class, defining fields like Parameter `MODULATION_AMPLITUDE` (integer valued, in dB), `MODULATION_PATTERN` (Value ROBOT, ALIEN, LUCY), and `Action START_MODULATION,STOP_MODULATION`.
There is no other modification required in the `MediaGroup` interface.
Note that the extensibility applies to

- Parameter (and Value)
- Action and Trigger
- EventType and Qualifier
- MediaErr

4.7 ResourceContainers: NetworkConnection, MixerAdapter, Mixer

`NetworkConnections` , `Mixers` and `MixerAdapters` are `ResourceContainers`, like the `MediaGroup`.
(`ResourceContainer` is a common ancestor class in the object design).

A `ResourceContainer` is described by a `MediaConfig` class. This class indicates which resources are included, and how they are arranged. `MediaConfigs` are non-mutable (read-only) objects.

A `MediaConfig` holds also default values for the various parameters of the resources. Note that `MediaConfig.getCustomizedClone(Parameters defaultValues)` can change those default values (but not the structure) and returns a new instance (the original cannot be changed).

4.7.1 `ResourceContainer` features

- `triggerAction`
- `getConfig()` returns a reference to the `MediaConfig` of the container.
- `setParameters` and `getParameters`
- `getURI` returns a URI that uniquely identifies this container (see below).
- *`release`* allows the application to indicate that the container and its resources can be discarded. Also, when a `MediaSession` is released, it releases all associated objects, i.e. objects created using the `createXXX` methods of this specific instance of `MediaSession`.

The last 3 features are actually inherited from MediaObject.

4.7.2 Referencing `MediaObjects`

Each `MediaObject` has a parent, which is its creator: for example, a `MediaGroup` is child of a `MediaSession`, which is child of a `MsControlFactory`.
As a result, the `MediaObjects` are placed in a tree structure, like files in a filesystem.

4.7.2.1 `MEDIAOBJECT_ID` attribute

Each `MediaObject` has an ID, unique in the scope of its parent (like a file name is unique in the scope of its directory).
The `MEDIAOBJECT_ID` is assigned be the system when the object is created, and cannot be changed afterwards.
Optionally, the application may provide its own ID, to replace the system-provided one:

```
Parameters params = myMediaSessionFactory.createParameters();
params.put(MediaObject.MEDIAOBJECT_ID, “MG.bridge-dialog-634”);
MediaGroup myMG =
myMediaSession.createMediaGroup(MediaGroup.PLAYER_RECORDER_SIGNALDETECTOR,
params);
```

Note: The ID is a non-mutable attribute, and implementations can base on this. Dynamic setting of ID is currently not allowed for simplicity.

4.7.2.2 `MediaObjects` have a unique URI

Each `MediaObject` can be designated by an URI, built by prepending the `MediaObject` ID with the URI of its parent. (This is similar to building a fullpath name with the file name and directory name).
The scheme for URIs is "**mscontrol://**". The upper part includes a reference to the media server, such as its IP address, and the ID of the `MediaSession` that created the object. For example:

```
mscontrol://16.16.184.23/ms45678/MediaGroup.mg9462
```

4.7.2.3 Abbreviated URI: omitting the mediaserver and mediasession

When the context is non-ambiguous, the application can omit the upper parts:

```
mscontrol:////MG.bridge-dialog-634
```

This is valid for referencing this `MediaGroup` to another object of the same `MediaSession`, like a `MediaMixer` (see the Video layout chapter).

4.7.2.4 Extended URI: designating media streams

An extended URI adds path components to designate the media streams of a `MediaObject`. For example:

mscontrol://16.16.234.47/MediaSession.ms12/Mixer.mx45/MixerAdapter.ma3/in/audio

designates the incoming audio stream of the `MixerAdapter` “MixerAdapter.ma3”.

5 Video Layout: Video Rendering, Video Conference

5.1 Summary

All video layout features are controlled by xml documents, currently SMIL is a proposed format.

A `VideoLayout` java object can be marshalled/unmarshalled from or to an xml document.

A `Mixer` can be configured to contain a `VideoRenderer` resource. This `VideoRenderer` can be given a `VideoLayout` to control the image that the mixer outputs.

5.2 Use of VideoLayout

The `VideoLayout` describes:

1. Regional definitions, using (x,y,z-index) positioning
2. Regional content, defined by URIs pointing to the media: (see below for the definition of media types)
    3. Discrete media: text, logos, etc, using their URI. As an exception, simple text can also be contained in-line instead of referenced through a URI. Use the *data:* scheme, as defined in RFC 2397.
    4. Continuous media: video clips, animated GIF
    5. Live media: defined by the URI of a `MixerAdapter`'s stream. (This where the xml document is linked to JSR 309 objects).

In the case of a SMIL layout, the regions are defined in the `<head>` of the document, while the contents are defined in the `<body>`.

5.3 Definitions

1. Live Media
   Media coming from a source external to the media server, that cannot be known in advance, cannot be repositioned (cannot be searched), with no applicable notion of duration.
   For example, a conference participant (represented by a `NetworkConnection`) is live media, as well as the feed coming from a RTSP server (represented by a `MediaGroup`), or real-time-text/chat coming from a remote agent.
2. Continuous Media
   Audio file, video file or other media for which there is a measurable and well-understood duration. For example, a five second audio clip is continuous media, because it has a well-understood duration of five seconds. Opposite of "discrete media".
3. Discrete Media
   Image file, text file or other media which has no obvious duration. For example, a JPEG image is generally considered discrete media, because there's nothing in the file indicating how long the JPEG should be displayed.

5.4 Examples

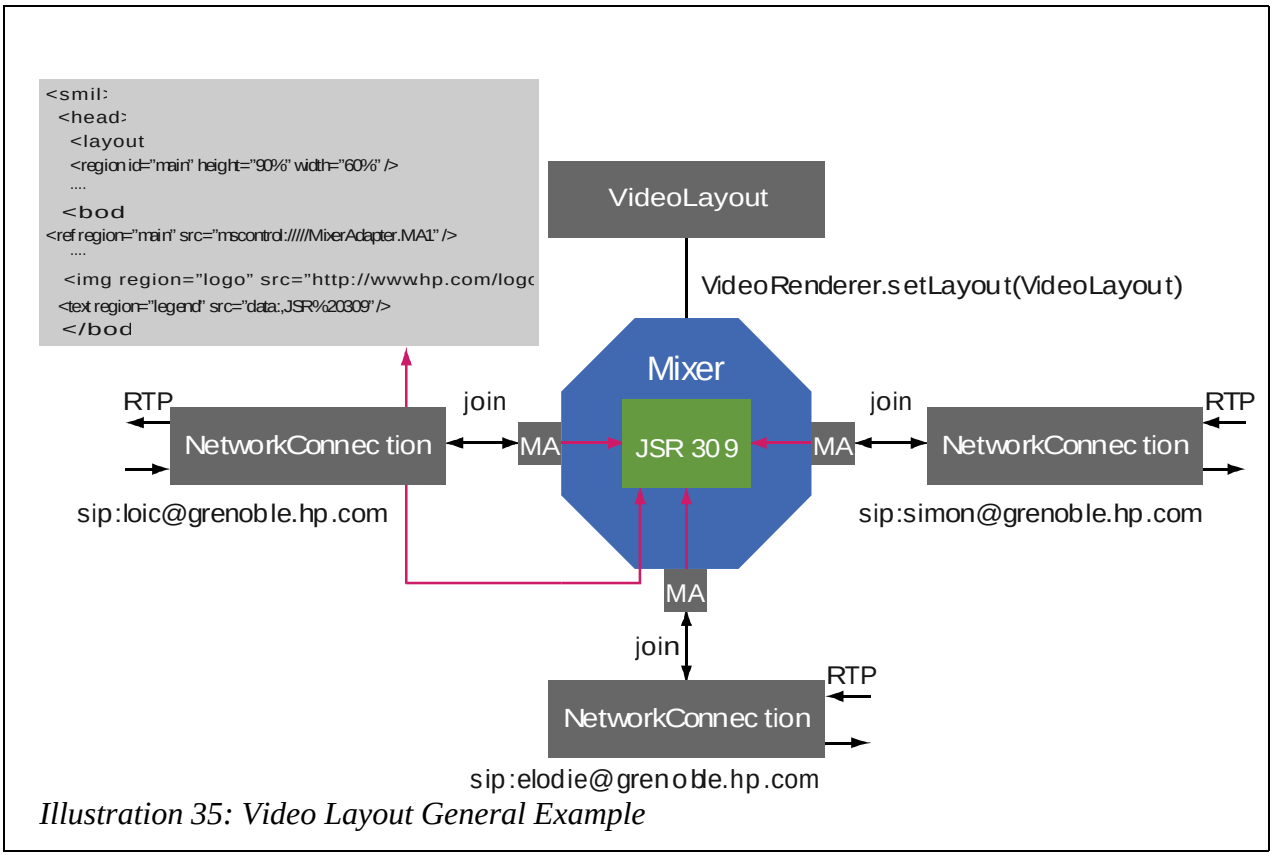


Illustration 35: Video Layout General Example

5.5 Per-Participant Layout

The same interface is available on a `MixerAdapter` object (representing a specific mixer port). This overrides the `VideoLayout` given to the `Mixer` (which applies by default to all particpants/ outputs), and makes it possible to specify a per-participant layout. The fact that the interface is on the `MixerAdapter` does not prevent the implementation to have the real job done by the `Mixer`, since a `MixerAdapter` is a slave object of its `Mixer`, with an unbreakable association. It is expected that implementations will optimize the layouts, i.e. if multiple `MixerAdapters` are given the same `VideoLayout`, only one renderer is used.

5.6 Designating a `MixerAdapter`'s stream in a VideoLayout

In all the xml formats envisioned to describe a layout, media is referenced by a URI. Therefore, a live stream will be referenced by the URI of its `MixerAdapter`.
Note that this URI does not depend on which object is joined to the `MixerAdapter`. The application is free to join and unjoin various `NetworkConnections`, `MediaGroups` and so on, without changing the layout, with the effect of changing the source of media that appears on the screen. This includes also pausing a `MediaGroup.play`, which should freeze the region, etc.

The format of a `MixerAdapter` stream URI is:

mscontrol://<appServerID>/<mediasessionID>/<MixerID>/<MixerAdapterID>/{in|out}/streamID

Notes:

- Since a `VideoLayout` is attached to a given mixer, the upper parts of the URI can be omitted in the layout document. For example: `mscontrol://///MixerAdapter.adapter3` is a valid URI string in an xml layout document.
- `streamID` is optional. When indicated, it is the string version of a member of the enum `JoinableStream.StreamType` (e.g.: `audio` or `video`).
  When the `streamID` is not indicated, the URI designates all the streams contained in the `MixerAdapter`.

Two operational modes are allowed. They are described below:

5.6.1 Building `MixerAdapters` First

The application sets up a `MediaMixer` and the required `MixerAdapters`, calls `MixerAdapter.getURI()` and then inserts the string representation(s) of the URI(s) in a (dynamically built) `VideoLayout` xml document.

5.6.2 Building the `VideoLayout` First

The application uses a static-defined xml document, possibly received from another system, such as a conference-policy server, and possibly written with offline content-generation tools.
The application now needs to have `MixerAdapters` with URIs that match those in the xml document. Note that only the `MixerAdapter` ID (the last part of the URI) is concerned.

This is achieved by assigning an ID when the `MixerAdapter` is created, see 3.7.2.

5.6.3 Virtual `MixerAdapter` URIs

It is useful to require that a region displays the loudest speaker. The application can do so by using a special, “virtual” URI, that the media server will interpret as “take the input of the most active talker, whoever this is”.

A closely related feature is the ability to require that one of the live streams is displayed, at the media server's choice (the application has no specific requirement about who this should be). Another “virtual” URI allows this.

Note that a virtual URI can represent a stream that is joined directly to the mixer, not necessarily through a `MixerAdapter`.

An URI that explicitly designates a given `MixerAdapter` is named an “explicit URI”, as opposed to “virtual URI”.

```
Interface VideoLayout {
```

```
  /**
   * Virtual URI, representing the stream of the most active participant.
   * If the Mixer cannot determine a most active, it should fall back to a
   * “anyActiveStream”.
   * This URI should never cause a duplicate display: if the most active
   * stream happens to be already explicitly displayed in an other region X,
   * then the media server should pick an alternative stream to display in X.
   */
  URI mostActiveStream;

  /**
   * Virtual URI, representing any stream that is active.
   * If the Mixer cannot determine an active stream, it should fall back to a
   * “anyStream”.
   * The exact selection is at the mixer's choice.
   * This URI should never cause a duplicate display: the most active stream
   * and the explicit URIs are valid candidates only if they are not already
   * displayed.
   */
  URI anyActiveStream;

  /**
   *  A virtual URI, representing one of the incoming streams, at the mixer's
   * choice.
   * This URI should never cause a duplicate display: the most active stream,
   * the anyActiveStream's and the explicit URIs are valid candidates only if
   * they are not already displayed.
   */
  URI anyStream;
}
```

5.7 Setting a VideoLayout on a Mixer

The `MediaMixerConfig` must include a `VideoRenderer` resource. The `VideoRenderer` interface has a `setLayout` method.

```
// Create a Mixer, using a predefined config that contains a video renderer
MediaMixer MyMixer =
mySession.createMediaMixer(MediaMixer.AUDIO_VIDEO_RENDERING);
// Get a handle on the VideoRenderer resource
VideoRenderer myRenderer = myMixer.getResource(VideoRenderer.class);
// Create a VideoLayout object, from inlined xml
VideoLayout myLayout = myMSFactory.createVideoLayout(“application/smil+xml”,
                                                     “<smil><head> ....”);
// Attach the layout to the renderer
myRenderer.setLayout(myLayout);
```

The interface is same for setting a `VideoLayout` on a `MixerAdapter`, except that the `VideoRenderer` resource must be included in the `MediaConfig` of the `MixerAdapter`.

5.7.1 Updating a `VideoLayout`

A `VideoLayout` is a non-mutable object (read-only). This allows caching, and avoids unbound effects at run-time.

As a result, any change in the layout requires to create another (modified) `VideoLayout`, and to call again `VideoRenderer.setLayout`.

It is implementation-dependent that the parts of the screen that are same as before, do not flicker.

5.7.2 Unsupported Layouts

The application will receive a *`VideoRenderingException`* (or one of its children) if the argument to *`VideoRenderer.setLayout`* is not usable by the renderer.

5.8 Code Sample

In this example, the layout has one main window and four smaller ones, plus a logo and a text region. The application wants the most active speaker (using VAD) to be displayed in the main window, the chairman of the conference in the upper small window, and other participants in the remaining 3 small windows. After some time, the application needs to show a video clip in the main window (content sharing).

```
// 1 - Regular video conferencing

MediaMixer myMixer = myMS.createMediaMixer(MediaMixer.AUDIO_VIDEO_RENDERER);

MixerAdapter chaimanAdapter = myMixer.createMixerAdapter(myMAConfig);
URI chairmanStream = chairmanAdapter.getURI();

myMixer.setLayout(myMSfact.createVideoLayout(“application/smil+xml”,
     "<smil>
       <head>
         <layout>
           <region id=\"main\" height=\"80%\" width=\"80%\" >
           <region id=\"thumb1\" top=\"0\"   height=\"20%\" left=\"80%\" >
           <region id=\"thumb2\" top=\"20%\" height=\"20%\" left=\"80%\" >
           <region id=\"thumb3\" top=\"40%\" height=\"20%\" left=\"80%\" >
           <region id=\"thumb4\" top=\"60%\" height=\"20%\" left=\"80%\" >
           <region id=\"logo\"   top=\"80%\" height=\"20%\" left=\"80%\" >
           <region id=\"caption\" top=\"80%\">
         </layout>
        </head>
        <body>
         <par>
          <ref region=\"main\" src=\"" + VideoLayout.mostActiveStream + "\" />
          <ref region=\"thumb1\" src=\"" + chairmanStream + "\" />
          <ref region=\"thumb2\" src=\"" + VideoLayout.anyStream + "\" />
          <ref region=\"thumb3\" src=\"" + VideoLayout.anyStream + "\" />
          <ref region=\"thumb4\" src=\"" + VideoLayout.anyStream + "\" />
          <img region=\"logo\" src=\"icons/HP.bmp\" />
          <text region=\"caption\" src=\"data:,JSR%20309%20Conference\" />
         </par>
        </body>
      </smil>");

// join the Chairman's NetworkConnection to the chairmanAdapter
// join other participants to the mixer, either through MixerAdapter or
directly.
```

```
....
// 2 - Contents sharing: the application wants to show a video clip to all
participants
MediaGroup myMediaGroup = myMS.createMediaGroup(MediaGroup.PLAYER);
MixerAdapter clipAdapter = myMixer.createMixerAdapter(myMAConfig);
URI clipStream = clipAdapter.getURI();
myMediaGroup.join(DUPLEX, clipAdapter);

// Change the layout so that the main window uses a definite MixerAdapter stream
myMixer.setLayout(myMSfact.createVideoLayout("application/smil+xml",
                        "<smil>
                             .... (same as above)
                             ....
                           <body>
                              <ref region=\"main\" src=\""+ clipStream + "\" />
                                  .... (same as above) ....
                              </body>");

myMediaGroup.play(URI.create("http://JSR309Presentation.3gp"), null, null);

// When chairman presses "3"
myMediaGroup.triggerAction(Player.PAUSE);
// When chairman presses "4"
myMediaGroup.triggerAction(Player.RESUME);
```

5.9 Preset Layouts

Some media servers support only a limited set of simple, traditional layouts. They can advertise what they support, using the method in `MediaSessionFactory`:

```
VideoLayout[] getPresetLayouts(int numberOfParticipants);
```

For each participant, the call returns the array of preset layouts suitable. (The array can have zero, one or multiple entries, depending on the media server versatility).

Usage:

```
// This application manages a 3-participants video conference.
// 1 – Create NetworkConnections, MediaMixer, possibly MixerAdapters
...
// 2 – Select a preset layout for 3 participants:
//     Take the first one proposed by the mediaserver (index 0)
VideoLayout threeGuys = myMSFactory.getPresetLayouts(3)[0];

// 3 – Use this layout as any other one:
VideoRenderer myRenderer = myMixer.getResource(VideoRenderer.class);
myRenderer.setLayout(threeGuys);

// 3 – join participants, etc
...
```

The application can display the preset layouts, in xml, to understand what they represent, and select the most appropriate for a given number of participants. This requires introspecting the xml document to understand how the regions are placed, etc.

Alternatively, the application can rely on standardized names, such as `xcon:dual-view-crop`, passed as the "type" argument of `getPresetLayouts(String type)`.

5.10 Per-Participant Data

Some applications need to display the name of each participant, in the bottom of their window or region. This can be done using the global mixer layout, by sub-dividing each participant window, and using the bottom sub-region. The application is responsible for maintaining a correct association between a name and a participant's stream. This is easy when the participants do not move, but harder when using VAD to delegate to the mixer the responsibility of displaying the most active speaker in a specific window – in this case, you need the name to follow the participant.

One solution for this is to proceed in two steps:

- Each participant is attached a personal, front-end `MediaMixer`, that builds a composite image containing their name in the bottom. This mixer is given a two-region layout (one for the participant's image, one for the name).
- The output of the front-end mixer is then fed into the main conference `MediaMixer`, just like a regular participant. The main mixer sees the image+name as a single stream, and switches them together.

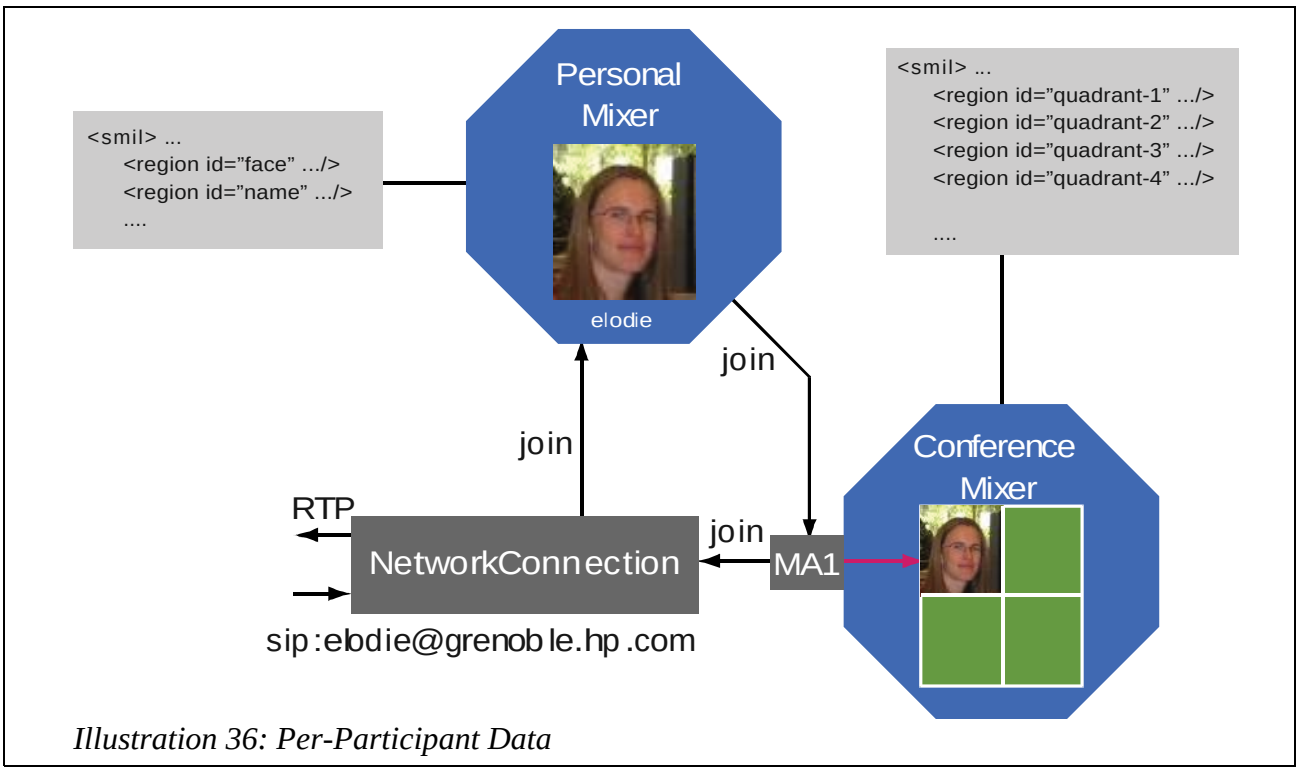


Illustration 36: Per-Participant Data

Note that the "personal mixer" concept is also useful for applications like video mail. In this case the mixer layout brings presentation and decoration to the (single) remote party.

5.11 End of Rendering

Some xml formats (like SMIL) include a notion of duration for the rendering. The duration can be related to discrete media (like a text that is displayed for a certain amount of time only), or the duration of continuous media.

The JSR 309 application is notified about the end of the rendering of the xml document, through the `EventNotifier` interface of the `VideoRenderer` class. This is the end of the full document; intermediate rendering events are not visible.

5.12 Adapting the Layout to the Number of Participants

In video conference applications, it may be desirable to adapt the layout so that the number of windows or regions corresponds to the number of connected participants.
The application must explicitly change the layout when a participant enters or leaves the conference. This cannot be delegated to the media server. The reason is that addition and removal of participants requires application processing, unlike the selection of the most active participant, which can be delegated.

6 NetworkConnection features

6.1 Codec Policy

6.1.1 Overview

The SdpPortManager is complemented with a feature to simplify the job of applications who want to influence on the codecs used in the network transport. The goal is that for the most usual actions, all what the application needs to know is the name of the codecs.
The application intention is expressed by verbs like “prefer”, “require”, “exclude”.
This is not tightly related to SDP and SDP negociation, and could be applied to other mecanisms for setting up streams.
Full SDP access is still provided on the SdpPortManager, but is required only for more advanced usage.

6.1.2 Application use-cases

A. I do not want to handle video, this is taking too much network bandwidth and/or I don't have video prompts.
B. I do not want to handle MPEG4 video or G.726 audio, because this application is not a “premium” service.
C. I do not want to handle outband DTMFs because some terminal equipments do not handle them correctly.
D. I can support only G.711- mulaw because I know that some equipments in my network support only this.
E. I can support any codec but I need outband DTMF support because some equipment does not process inband DTMFs
F. If I am offered both G.711 and G.729, I choose G.711 because I have enough network bandwidth and it saves cpu cycles.

6.1.3 Codec status

Each codec has a status, among the following:

- *allowed*
- *required*
- *excluded*

The association of a status to a codec is defining a “Codec Policy”. This policy will be applied when the connections is setup.
The connection setup will fail if:

- an *excluded* codec must be used (or else, the media-type would have no available codec)
- a *required* codec is not available

Some cases are not considered as an error by the JSR 309 spec, but may yet cause a setup failure from the media server or from the user agent:

- Excluding both G.711 and telephone-event (some advanced codecs cannot encode inband

DTMFs reliably)
- The only remaining codec is telephone-event

On top of this, the application can indicate preferences, by an ordered list of codecs. The preferences are applied after all other constraints have been applied, and never cause a failure.

6.1.3.1 Media types

Media types can be excluded.
The setup will fail if all the available media-types are excluded by the policy.

6.1.4 CodecPolicy interface

Initially all codecs have the *allowed* status.
The interface proposes 5 methods to change this:

1. **setMediaTypeCapabilities(String mediaType)** to exclude video for example.
   This covers use-case A, with setMediaTypeCapabilities("audio");

2. **setExcludedCodecs(String[] codecs)**
   use-case B: setExcludedCodecs(new String[] {"MPEG4", "G726"});
   use-case C: setExcludedCodecs(new String[] {"telephone-event"});

3. **setCodecCapabilities(String[] codecs)**
   All codecs *not in the array* are set to "excluded".
   Use-case D: setCodecCapabilities(new String[] {"PCMU","PCMA", "telephone-event"}

4. **setRequiredCodecs(String[] codecs)**
   use-case E : setRequiredCodecs(new String[] {"telephone-event"}

5. **setCodecPreferences(String[] codecs)**
   This method indicates only a preference. The order of the preference is the order of the array. Codecs that are not listed have a still lower preference.
   Never causes a failure.
   use-case F: setCodecPreferences(new String[] {"PCMU","PCMA"});

All method calls *override* the previous calls of the same method (do not accumulate individual codec information).
The corresponding getters are provided as well.

6.1.5 Implementation guidelines

This chapter is focusing on the case of a SdpPortManager.

6.1.5.1 Recall on SDP negociation

1. One of the call participants (UserAgent, MediaServer) makes an offer. The fact that the

offerer is the User Agent or the MediaServer is **not** related to concepts like incoming/outgoing call.

2. The other participant makes an answer, which must be a subset of the offer.
3. The offer may contain multiple codecs, in this case they are ordered by preference of the offerer.
4. The answer may contain multiple codecs for a given media type (e.g., audio), in this case the participants may dynamically switch between codecs (switch between RTP payload) during the call. Outband DTMFs, using RFC2833 packets are an example of "dual" codec.
5. The "negociation" is only the possibility for the answerer to **remove** codecs from the offer, or to re-order them. More complex features need multiple offer-answer exchanges, each of them based on those principles.

6.1.5.2 Algorithm

The Codec Policy can be enforced by filtering/reordering the offer before it is proposed to the other party (whoever the offerer is).

The filtering/reordering will happen at two points:

- When the application has called *generateSdpOffer()*, filter the media server offer before sending the OFFER_GENERATED event.
- When the application calls *processSdpOffer(byte[] offer)*, filter the argument offer before forwarding to the media server.

In the failure cases specified in chapter 3, SDP_NOT_ACCEPTABLE will be set in the completion event.

6.1.5.3 Setting CodecPolicy on SdpPortManager

An implementation must not assume that the provided `CodecPolicy` will not change after it is set and thus, when `SdpPortManager.setCodecPolicy` is called, the implementation should make a deep copy of the `CodecPolicy` object before using the object in any way, including making any sanity checks on the object. The implementation should make sanity checks on the object when `SdpPortManager.setCodecPolicy` is called, after it has copied the `CodecPolicy`.

For the same reason, `getCodecPolicy` should return a copy, and not the working object of the `SdpPortManager`.

A new `CodecPolicy` can be based on an existing `CodecPolicy` as follows:

```
CodecPolicy policy = mySdpPortManager.getCodecPolicy();
policy.setExcludedCodecs(....);
mySdpPortManager.setCodecPolicy(policy)
```

The last call installs the new `CodecPolicy` which overwrites the previous one, providing the new `CodecPolicy` passes sanity checks.

6.1.6 Examples

In the table below, the first row shows the “initial offer”, and each cell shows the “filtered offer” that will be sent to the answerer, after applying the methods in the first column.

Initial offer • applied methods | V	m=audio a=rtpmap:0 PCMU a=rtpmap:8 PCMA a=rtpmap:97 iLBC m=video a=rtpmap:31 H261 a=rtpmap:32 MPV	m=audio a=rtpmap:0 PCMU	m=audio a=rtpmap:45 G726 a=rtpmap:97 iLBC a=rtpmap:99 tel-event m=video a=rtpmap:31 MPV a=rtpmap:32 H261	m=audio a=rtpmap:45 G726 a=rtpmap:97 iLBC a=rtpmap:0 PCMU a=rtpmap:8 PCMA a=rtpmap:99 tel-event
setMediaTypeCapabilities(“audio”)	m=audio a=rtpmap:0 PCMU a=rtpmap:8 PCMA a=rtpmap:97 iLBC	unchanged	m=audio a=rtpmap:45 G726 a=rtpmap:97 iLBC a=rtpmap:99 tel-event	unchanged
setCodecPreferences(“PCMA”, ”H263”, ”H261”, ”PCMU”,)	m=audio a=rtpmap:8 PCMA a=rtpmap:0 PCMU a=rtpmap:97 iLBC m=video a=rtpmap:31 H261 a=rtpmap:32 MPV	unchanged	unchanged	m=audio a=rtpmap:8 PCMA a=rtpmap:0 PCMU a=rtpmap:45 G726 a=rtpmap:97 iLBC a=rtpmap:99 tel-event
setRequiredCodecs(“telephone-event”)	failure	failure	unchanged	unchanged
setCodecCapabilities(“PCMU”)	m=audio a=rtpmap:0 PCMU m=video a=rtpmap:31 H261 a=rtpmap:32 MPV	unchanged	failure	failure
setExcludedCodecs(“telephone-event”)	unchanged	unchanged	m=audio a=rtpmap:45 G726 a=rtpmap:97 iLBC m=video a=rtpmap:31 MPV a=rtpmap:32 H261	m=audio a=rtpmap:45 G726 a=rtpmap:97 iLBC a=rtpmap:0 PCMU a=rtpmap:8 PCMA
setExcludedCodecs(“PCMU”)	m=audio a=rtpmap:8 PCMA a=rtpmap:97 iLBC m=video a=rtpmap:31 H261 a=rtpmap:32 MPV	failure	unchanged	m=audio a=rtpmap:45 G726 a=rtpmap:97 iLBC a=rtpmap:8 PCMA a=rtpmap:99 tel-event
setExcludedCodecs(“H261”, “MPV”)	failure	unchanged	failure	unchanged
setMediaTypeCapabilities(“video”)	m=video a=rtpmap:31 H261 a=rtpmap:32 MPV	failure	m=video a=rtpmap:31 MPV a=rtpmap:32 H261	failure

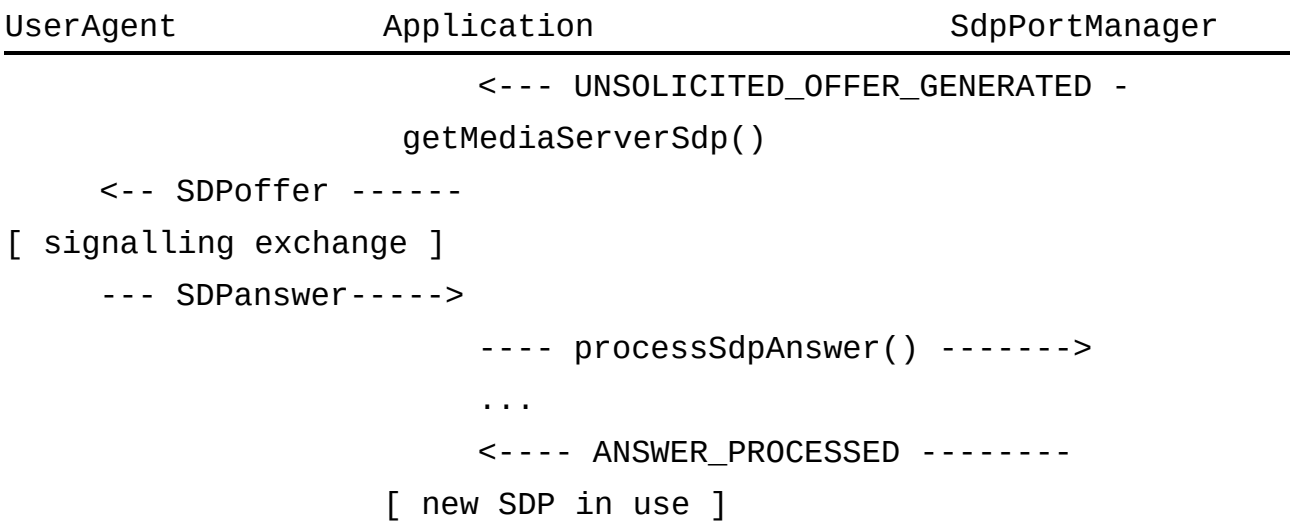
6.2 SDP negociation

A media connection is determined by a pair of Session Descriptions, as described in the `SdpPortManager` javadoc: the MediaServer and the UserAgent SDP.

Before the offer-answer exchange is complete, both SDP are considered “tentative”. For this reason, the `SdpPortManager.getMediaServerSessionDescription and getUserAgentSessionDescription` methods return null, or return the previously agreed SDP. The application must keep track of the tentative SDP, that it gets either from the signalling interface, or from the `SdpPortManagerEvent.getMediaServerSdp` method.

In most cases the application is at the origin of the SDP negociation, associated to a signalling operation from/to the UserAgent.

It may happen that the Media Server requests a change of SDP. The application is notified of this by the `UNSOLICITED_OFFER_GENERATED` event. The offer from the Media Server is obtained by `getMediaServerSdp.` Until the application calls processSdpAnswer, the SDPs in use are unchanged, and can be compared to the new offer by calling `getMediaServerSessionDescription` or `getUserAgentSessionDescription`.



If the offer fails, the SDPs remain unchanged.

Notes on Joining the `NetworkConnection`:

There is no restriction on the order between joining a `NetworkConnection` and setting up its SDP. There are some cases where it must be possible to join the `NetworkConnection` before the actual SDP is fully specified, for instance:

- in order to ensure reception of the first media packet, like a video I-Frame, it is good practice for the Application to first join the `NetworkConnection` and the `Mixer` before opening up the SDP on the network in order to enable reception from the other end. In this case the SDP negotiation (e.g. `SdpPortManager.processSdpOffer`), would be

initiated after the join.

- it is always possible for the Application, or the Media Server to renegotiate the SDP after the `NetworkConnection` has been joined, again the join topology must support changes in the SDP and an implementation must return appropriate errors if either an operation related to SDP negotiation or a join operation cannot complete due to transcoding errors or other combinations of join topology and SDP values that the Media Server cannot actually support.

The join topology must be separated from the actual flow of Media, they are independent and controlled by different API features. Joining objects is different function than allowing media to flow on these links. Two objects can be joined while the media is not flowing yet, e.g. you have not enabled a Layout on the `Mixer` or the `NetworkConnection` is still in SDP mute mode.

6.3 Advanced Features: DTMFs carried in SIP-INFO and VFU Requests

6.3.1 Using `NetworkConnections` and `MediaGroups` Together

The `MediaGroup` is the object used by an application to perform the desired media interaction: play files, receive signals, etc. Such applications need to be as independent as possible from the media transport - not tied to RTP or TDM, SIP or ISUP, etc. They also need the same interface to interact with a single user, or with multiple users (playing the same file to multiple users), or with a conference (a group of users). VoiceXML applications use a `VxmlDialog`, which can be viewed as an enhanced `MediaGroup`.

On the other hand, the `NetworkConnection` object handles everything specific to a remote agent, including all transport-specific features. Part of an application will be dedicated to driving the `NetworkConnection`, in association with the signaling interface and the media transport interface. The rest of the application will be network-independent.

6.3.2 Reception of DTMFs Carried by an RTP Stream

According to the standards, in-bound DTMFs are transported with the media stream, and out-bound as special RTP packets (outband).

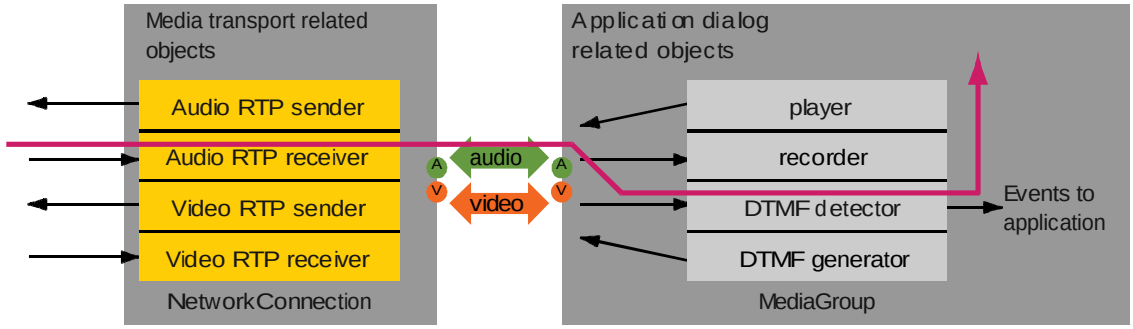


Illustration 37: DTMF Reception

A DTMF is typically received on the audio RTP port, and presented to the application in the `SignalDetector` of the `MediaGroup`. The only relationship between the media-transport-related code and the application dialog, is through the `join` between `NetworkConnection` and `MediaGroup`/`VxmlDialog`.

6.3.3 DTMFs Carried in SIP INFO Messages

When a DTMF is received as a payload in a SIP-INFO message, the application can explicitly inject the event into the dialog-related part using a special channel:

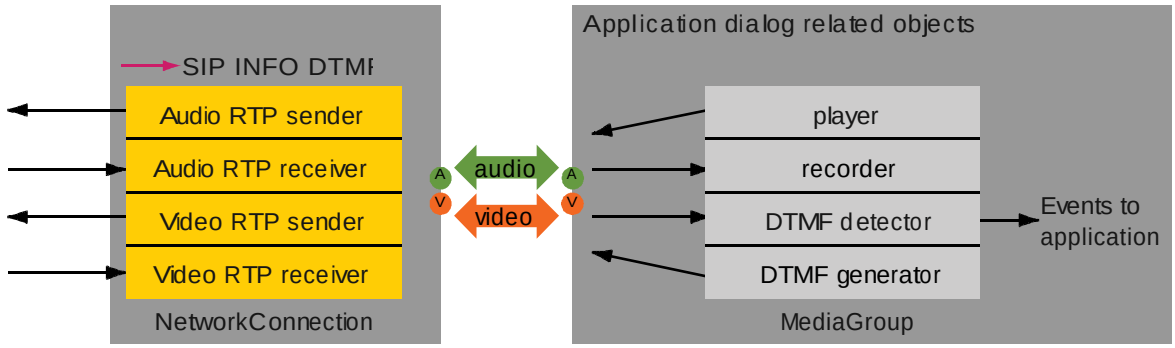


Illustration 38: DTMF in SIPInfo Messages

This channel is different for each dialog type. `VxmlDialog`, requires the use of ad-hoc events. It is hard to make sure that the final result is identical, regardless of the way the DTMF was transported. Note also that multiparty applications need to support mixed mode, where some participants send DTMFs in the usual way, and some others using SIP INFO.

This means that this solution breaks application transport-independence. It also adds a data path parallel to the main path established by the `join` calls, thus breaking the media composition scheme of JSR 309.

An alternative is to constrain the issue in the media transport related part, by re-injecting a DTMF signal in the `NetworkConnection`:

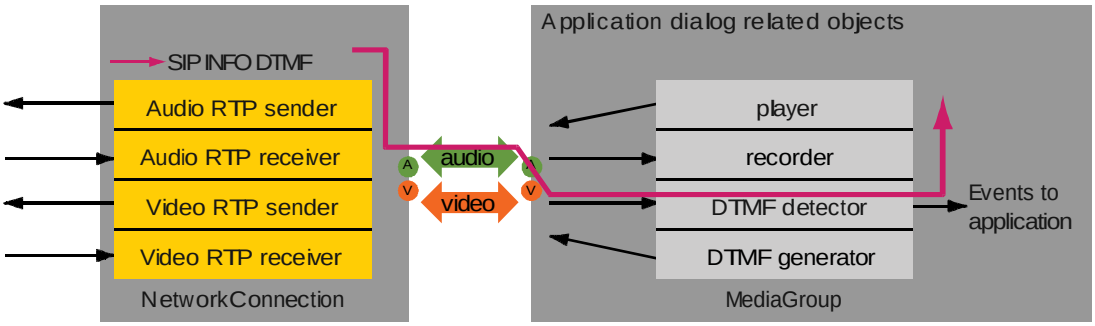


Illustration 39: DTMF Signal in NetworkConnection

In this situation all the dialog-related parts of the application see all DTMFs in the same way.

To support this, the application needs to use a `SignalGenerator` interface to the `NetworkConnection` (the same interface as the `MediaGroup`). The signaling-related part of the application catches incoming SIP INFOs, and calls the `SignalGenerator` accordingly.

Note that the `SignalGenerator` of the `NetworkConnection` sends signals into the system (into the media server), and not out to the network, as you might think intuitively.

A `SignalDetector` interface allows the reverse transformation. A DTMF generated by the application is intercepted in the `NetworkConnection` by the signaling-related code, and transformed into an outgoing SIP INFO message. If the same DTMF is sent to multiple participants, each signaling-related code will handle it separately as needed, thus providing support for mixed-mode applications.

A few applications generate DTMFs, however this applies also to bridging NetworkConnections in mixed mode.

6.3.4 Video Fast Update

Video Fast Update requests are needed when an object starts receiving a video stream. The key frames (needed to construct a full image) can be several seconds apart, this means that the receivers sees a partial image as they arrive. This occurs notably when starting a record operation, and when accepting a new participant in a video conference.

VFUs have no application semantic, they are used for fine-grain tuning to optimize the image quality. The application does not interact with VFUs.

There are two ways of transporting `VFURequests`: SIP INFO messages, and RTCP packets.

6.3.4.1 VFU Requests with RTCP Transport

Using RTCP packets, `VFURequests` can be handled completely transparently by the media-server. The Recorder will makes sure that the `NetworkConnection` sends out a `VFURequest` RTCP packet (and so will the Mixer). If a `VFURequest` is received by a `NetworkConnection`, it is forwarded in the media server objects, until it reaches an object capable of generating a key-frame (a player, or a mixer), or until it reaches an outside boundary of the system, that is another `NetworkConnection`.

Note: A `VFURequest` goes “upstream”, in the reverse direction from the stream: a play-only application may receive a `VFURequest` from the receiving UA. At the junction of the `NetworkConnection` and the `MediaGroup`, it moves in the recorder's direction:

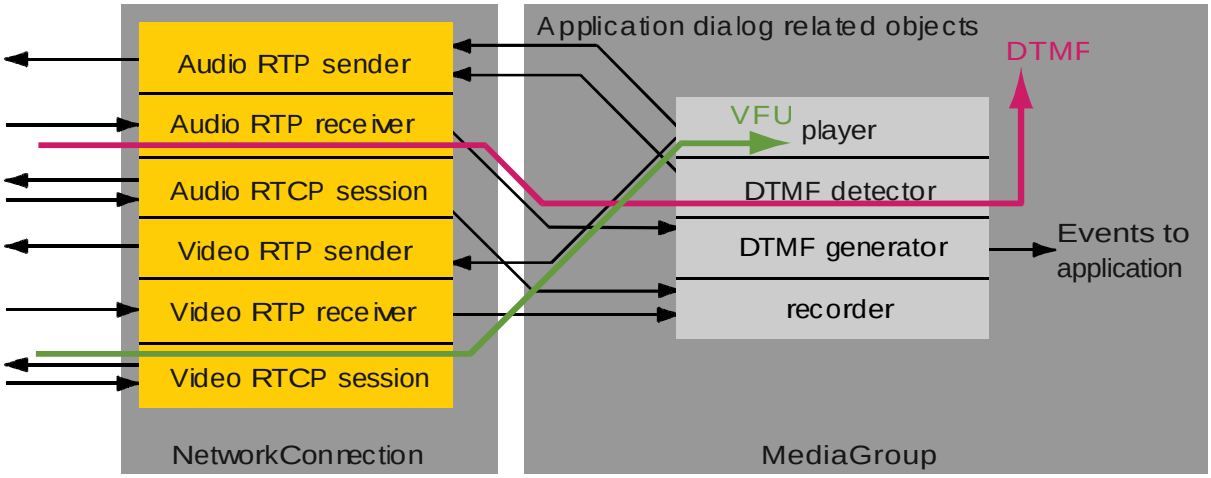


Illustration 40: VFURequest with RTCP Transport

6.3.4.2 VFU requests carried in SIP INFO

Using `VFURequests` in SIP INFO, means dealing with the same issue as using DTMFs in SIP INFO: transporting a mid-call media information in signaling, creating coupling between signaling and media layers.

This proposal adds a new `VFURequest` signal, that can be handled by the network-related part of the application. The `VFURequest` signal complements the list of `DMTF_0, _1, ... _9, _A, _B, _C, FaxTone, VFURequest`. The handling of `VFURequest-in-SIP-INFO` becomes a special use-case of the `DTMF-in-SIP-INFO` paragraph above.

Note: When the application calls the `SignalGenerator` of the `NetworkConnection` to send a DTMF (received by SIP INFO) “inside the media server”, the message is sent to whatever object receives data from the `NetworkConnection` (e.g., the DTMF detector in a `MediaGroup`, as shown in fig 3 above). There may be multiple such recipients.

As mentioned in paragraph 4.1, VFUs are transferred upstream. When an application calls the SignalGenerator of the NetworkConnection to send a VFURequest “inside the media server”, the request is transmitted the upstream object playing the `NetworkConnection`. There will be only one such object, since multiple sources are not allowed by `join`.

7 Service Provider Interface

This JSR defines a Service Provider Interface (SPI), which any implementation of the Media Server Control API must implement. The Service Provider Interface, which any implementer must implement is:

•javax.media.mscontrol.spi.Driver, interface that every Media Server Control API driver must implement. Each driver should supply a class that implements this interface. This interface is used by the DriverManager to retrieve MsControlFactory instances, and to retrieve information on supported properties for MsControlFactory instances of a certain driver.

Another important class of the javax.media.mscontrol.spi package is:

•javax.media.mscontrol.spi.DriverManager, a singleton class, which drivers register themselves in, to make them available for consumption by applications. Applications retrieves MsControlFactory instances through the DriverManager.

The Service Provider Interface isolates the application from driver configuration, improving portability of applications, and makes it possible to co-deploy several versions of a specific driver (same vendor).

The Service Provider Interface decouples the driver and the execution environment, such as an J(2)EE container, not requiring the execution environment to be aware of Media Server Control API drivers.

7.1 Installing/Loading drivers

For portability reasons and to ease configuration of MsControlFactory instances, all deployed drivers must register in the DriverManager. Depending on packaging and deployment, a driver will register in the DriverManager either explicitly or implicitly. The following sections describe different ways of packaging and deployment of drivers.

7.1.1 Service Provider JAR

As part of its initialization, the DriverManager, will load all drivers packages as jar files including a file META-INF/services/javax.media.mscontrol.spi.Driver. This file contains the name of the driver's implementation of javax.media.mscontrol.spi.Driver. Drivers packaged this way does not have to explicitly register in the DriverManager.

See http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html#Service Provider for more information on Service Providers.

To ensure drivers can be loaded using this mechanism, drivers are required to provide a no-argument constructor.

7.1.2 Alternative packaging

Drivers not packaged as a jar file, must ensure they register in the DriverManager upon initialization**.** Drivers implemented as a J(2)EE application/module such as a SIP Servlet application or a resource adapter (JCA), will not be registered automatically as the DriverManager class is loaded by the JVM, and therefore must explicitly register themselves in the DriverManager.

7.1.3 Deploying several versions of a specific driver

Only one version of a class can be loaded by a single class loader at any time, hence it is not possible to co-deploy two versions of a driver as long as they are using the same full class name. Drivers packaged as jar files and loaded by the DriverManager would normally be loaded by the system class loader, and accordingly only one version of the driver would be accessible by applications. Drivers implemented as J(2)EE applications/modules would normally be loaded by different class loaders, hence several versions of a driver could actually be co-deployed.

7.2 JNDI and Dependency injection

7.2.1 JNDI binding of MsControlFactory

In certain execution environments, binding MsControlFactory instances in the global JNDI tree are preferable as that allows one to isolate Driver/MsControlFactory configuration from the application.

What entity/component that creates, configures and binds MsControlFactory in the global JNDI tree is intentionally unspecified. This could for instance, be done by a driver upon initialization or by a J(2)EE container during startup.

7.2.2 Dependency injection

In execution environments supporting dependency injection, MsControlFactory instances can preferable be injected using @Resource annotations. Whether or not a MsControlFactory instance can be injected into a certain class, such as a javax.servlet.Servlet, is to be specified by every execution environment supporting the @Resource annotation.

7.3 Naming conventions

7.3.1 Driver name

The name a driver registers in the DriverManager, must uniquely identity the driver.

7.3.2 Recommendation on JNDI names

It is recommended that MsControlFactory instances that are bind in the global JNDI tree, use the following JNDI name format:

mscontrol/<unique-driver-name>/<unique instance identifier>

For instance:

mscontrol/vendorA_1.0.1/ms1

7.4 Retrieving MsControlFactory

An application can retrieve an instance of MsControlFactory by the following means:

•Through the DriverManager:

```
DriverManager.getFactory(“SampleDriver_1.0.1”, myProperties);
```

•Through JNDI lookup (global JNDI):

```
(MsControlFactory) ctx.lookup(“mscontrol/sample_1.0.1/ms1”);
```

•Through dependency injection:

```
@Resource(name=“mscontrol/sample_1.0.1/ms1”)
Private MsControlFactory factory;
```

7.4.1 Example: Retrieving MsControlFactory instances through DriverManager

An application is accessing three different media servers, two of vendor A, and one of vendor B. To create media sessions associated with each of these media servers, three different MsControlFactory instances would be needed.

Using the DriverManager, these media sessions could be created as shown below:

```
//create media session associated with MS of vendor A and host ms1.example.com
Properties props1 = new Properties();
props1.setProperty(MEDIA_SERVER_URI, “sip:ms1.example.com”);
MsControlFactory mscFactory1 = DriverManager.getFactory(“vendorA_1.0.1”,
props1);
MediaSession session1 = mscFactory1.createMediaSession();

//create media session associated with MS of vendor A and host ms2.example.com
Properties props2 = new Properties();
props1.setProperty(MEDIA_SERVER_URI, “sip:ms2.example.com”);
MsControlFactory mscFactory2 = DriverManager.getFactory(“vendorA_1.0.1”,
props2);
MediaSession session2 = mscFactory2.createMediaSession();

//create media session associated with MS of vendor B and host ms3.example.com
Properties props3 = new Properties();
props1.setProperty(MEDIA_SERVER_URI, “sip:ms3.example.com”);
MsControlFactory mscFactory3 = DriverManager.getFactory(“vendorB_1.0.1”,
props3);
MediaSession session3 = mscFactory3.createMediaSession();
```

mscFactory1 and mscFactory2, would belong and be created by the same driver instance.

7.4.2 Example: Retrieving MsControlFactory instances using dependency injection

An application is accessing three different media servers, two of vendor A, and one of vendor B. To create media sessions associated with each of these media servers, three different MsControlFactory instances would be needed.

Using JNDI lookup or dependency injection, one would first have to create and bind three different instances of MsControlFactory in the global JNDI tree to make them available to applications. In a managed execution environment like a J(2)EE application server or a standalone container, this could be done upon server startup.

Provided all needed MsControlFactory instances are available in the global JNDI tree, the application would retrieve them as shown below:

```
@Resource(name="mscontrol/vendorA_1.0.1/ms1")
private MsControlFactory mscFactory1;
@Resource(name="mscontrol/vendorA_1.0.1/ms2")
private MsControlFactory mscFactory2;
@Resource(name="mscontrol/vendorB_1.0.1/ms3")
private MsControlFactory mscFactory3;
```

And the media sessions would be created as shown below:

```
MediaSession session1 = mscFactory1.createMediaSession();
MediaSession session2 = mscFactory2.createMediaSession();
MediaSession session3 = mscFactory3.createMediaSession();
```

In contrast to the example using the DriverManager for retrieving MsControlFactory instances above, in this example all driver configuration except the JNDI names are separated from the application.

7.5 Driver Configuration

This section describes how one could create and configure MsControlFactory instances, and how to make configuration information available to GUI-based management tools such as web-based management consoles.

The javax.media.mscontrol.spi.Driver interface, defines a method getFactoryPropertyInfo(), which returns information about supported properties for MsControlFactory instances of a driver. There is also a getProperties() method defined in the MsControlFactory interface, which returns a read-only copy of a MsControlFactory instance's properties.

The javax.media.mscontrol.spi.DriverManager class defines a getDrivers() method, which returns a list of all available drivers.

By using the above methods, it is possible for a generic GUI tool to create, configure and bind MsControlFactory instances in the global JNDI tree. In addition, it is also possible to list the properties of existing factory instances by using the MsControlFactory.getProperties() method.

A possible sequence of operations for creating a MsControlFactory instance and binding it in the global JNDI tree using a GUI tool is given below:

1.Invoke DriverManager.getDrivers() to provide a list of all available drivers.

2.Select a driver by specifying a driver name from the list.

3.Invoke Driver.getFactoryPropertyInfo() to find out which properties this driver supports and requires.

4.Set the properties.

5.Invoke DriverManager.getFactory(String, Properties) with the specified properties as argument, to get hold of a factory instance.

6.Choose a global JNDI name for this factory

7.Bind the factory in the global JNDI tree

The only MsControlFactory property specified in the Media Server Control API is MEDIA_SERVER_URI, which defines the URI of a media server, e.g. sip:ms@192.168.1.2:5060.

8 Feature Discovery by the Application

8.1.1 `SupportedFeatures` interface

Each `MediaConfig` (the template describing the resources in a container) can tell what features it supports: the application can call `MediaConfig.getSupportedFeatures()` and process the returned Sets of features.

For example, to know if the `MediaGroup.PLAYER_RECORDER_SIGNALDETECTOR` contains a `SpeechDetector` (to trim the beginning and end silences in a recording):

```
MediaConfig ivrConfig =
myMediaSessionFactory.getMediaConfig(MediaConfig.PLAYER_RECORDER_SIGNALDETECTOR)
;
Set<Parameter> supportedParams =
ivrConfig.getSupportedFeatures().getSupportedParameters();

if(supportedParams.contains(SpeechDetectorConstants.INITIAL_TIMEOUT)) {
    // we can trim the initial silence
} else {
    // No silence detection on this media server
}
```

Note that this piece of code does not need to instantiate a `MediaGroup`, and can be run in the initialization code of the application.

Another important feature is video support. This can be queried using `MediaConfig.`**`hasStream`**(StreamType atype). In the example above:

```
if (ivrConfig.hasStream(StreamType.video))
   // video is supported
```

8.1.2 ResourceContainer Instances

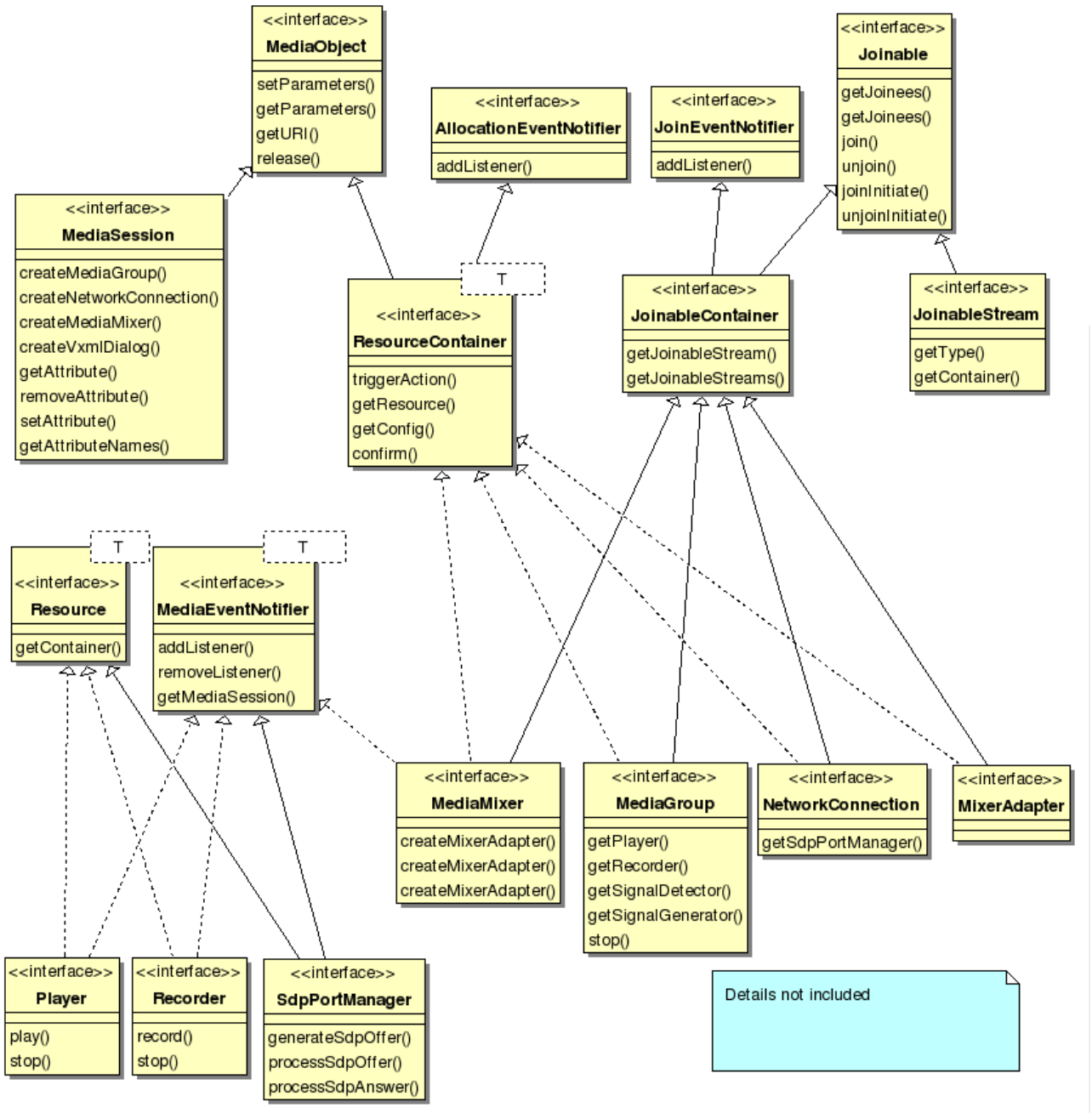
A given instance of `ResourceContainer` can be queried at run time using the `getConfig()` method to access the `MediaConfig`:

```
if (myMediaGroup.getConfig().hasStream(StreamType.video))
   // video is supported by myMediaGroup
```

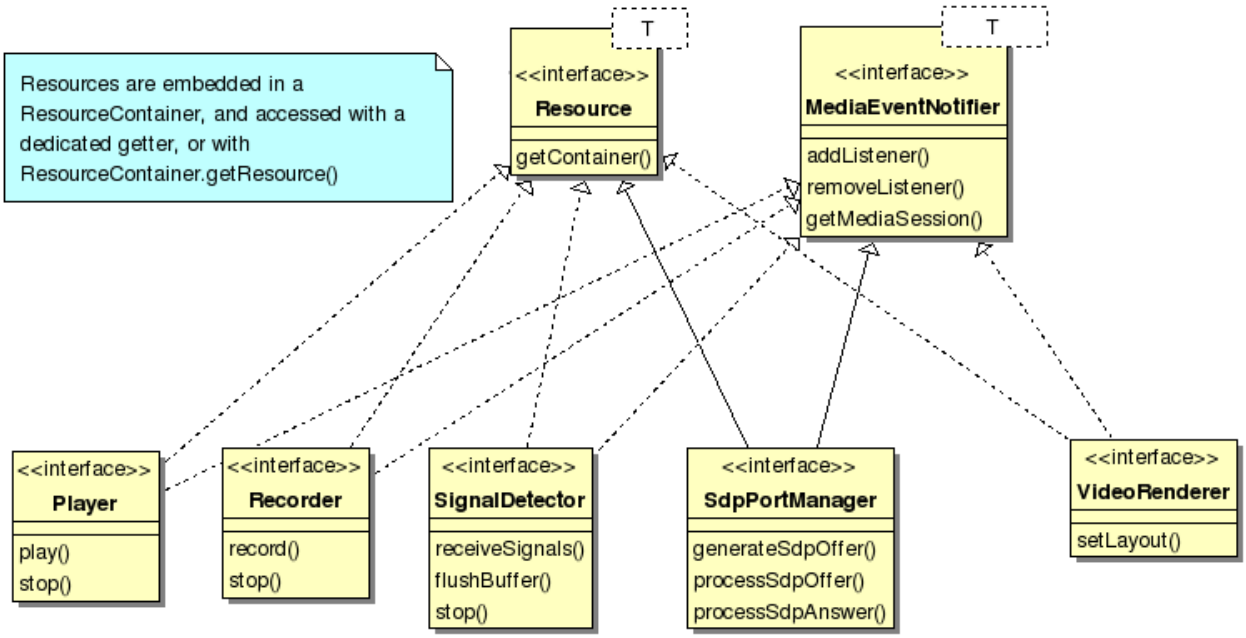
9 Class Diagrams

The following simplified class diagrams are intended to help understand the object model.

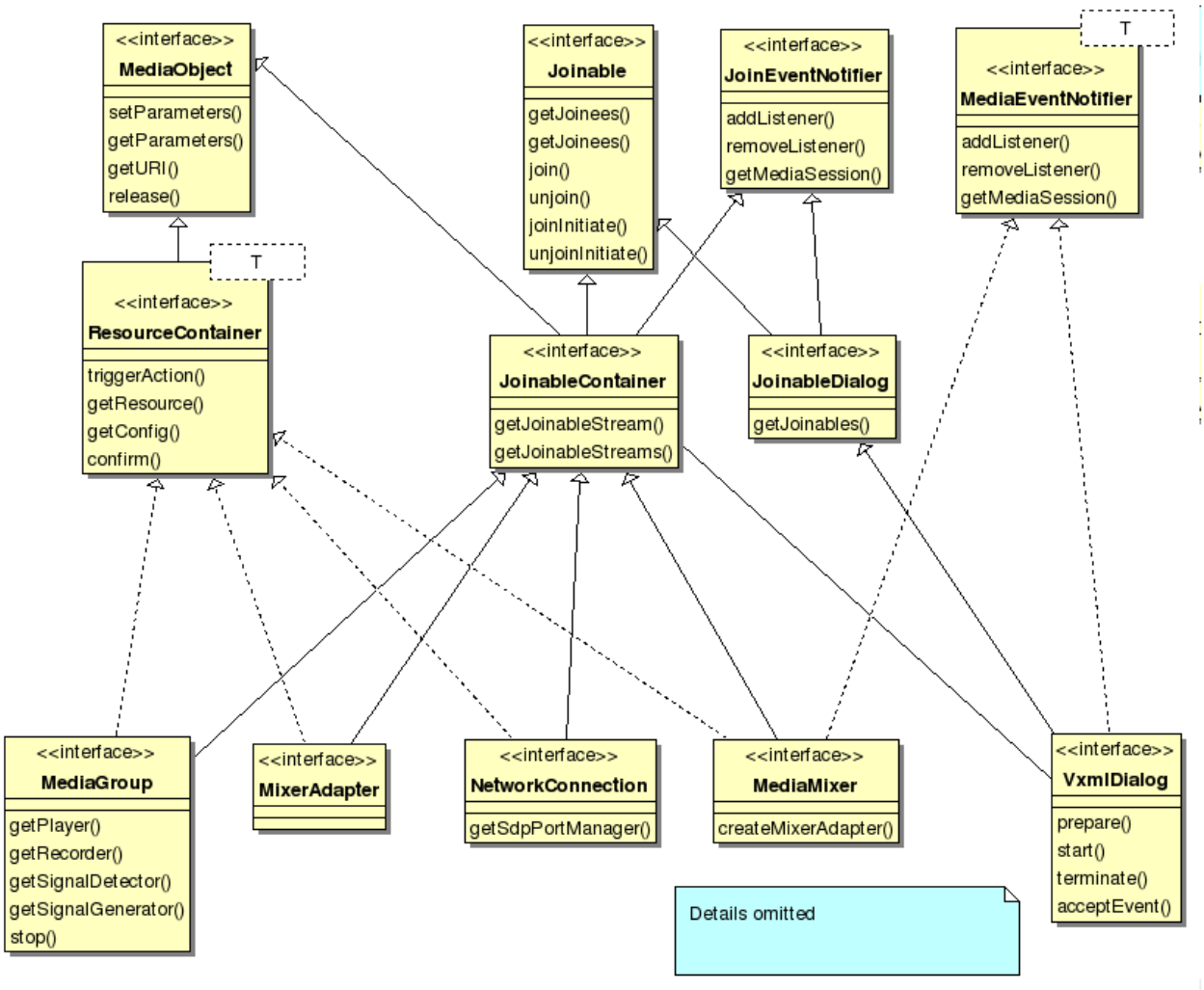
9.1 Overview



9.2 Resources



9.3 Containers



9.4 Events