Media Sessions Markup Language (MSML)
draft-melanchuk-sipping-msml-06

Abstract

The Media Sessions Markup Language (MSML) is used to control and invoke many different types of services on IP media servers. Clients can use it to define how multimedia sessions interact on a media server and to apply services to individual or groups of users. MSML can be used, for example, to control media server conferencing features such as video layout and audio mixing, create sidebar conferences or personal mixes, and set the properties of media streams. As well, clients can use MSML with other languages such as the Media Objects
Markup Language (MOML) or VoiceXML to interact with individual users or with groups of conference participants.

Table of Contents

1. Introduction ................................................. 4
2. Conventions used in this document .......................... 5
3. Glossary .................................................. 6
4. MSML SIP Usage ............................................ 8
5. Language Structure ........................................... 10
  5.1. Package Scheme ......................................... 10
  5.2. Profile Scheme ......................................... 10
  5.3. Execution Flow ......................................... 11
  5.4. MSML Root Element ....................................... 12
  5.5. Sending Events to a Media Server ......................... 13
  5.5.1. <send> ............................................ 13
  5.6. Transaction Results and Notifications ..................... 13
  5.6.1. <result> ........................................... 13
  5.6.2. <event> ............................................. 14
6. Media Server Object Model .................................... 15
  6.1. Objects ................................................. 15
  6.2. Identifiers .............................................. 18
7. Media Streams ................................................ 21
  7.1. <join> ................................................ 22
  7.2. <modifystream> ......................................... 24
  7.3. <unjoin> ............................................... 25
  7.4. <stream> Elements ....................................... 26
    7.4.1. Audio Stream Properties ............................ 27
    7.4.2. Video Stream Properties ............................ 28
  7.5. <monitor> ............................................... 29
8. Conferences .................................................. 30
  8.1. <createconference> ...................................... 30
  8.2. <modifyconference> ...................................... 32
  8.3. <destroyconference> ..................................... 33
  8.4. Audio Mix .............................................. 34
    8.4.1. N-Loudest ......................................... 34
    8.4.2. Active Speaker Notification ........................ 34
  8.5. Video Layout ............................................ 35
    8.5.1. Root Layout ........................................ 35
    8.5.2. Regions ............................................ 36
    8.5.3. Stream Selection ................................... 40
  8.6. Reserving Conference Resources .......................... 42
9. Dialogs ...................................................... 43
  9.1. <dialogstart> .......................................... 43
  9.2. <dialogend> ............................................ 44
10. Response Codes .............................................. 46
11. Examples ................................................... 48
1. Introduction

Media servers contain dynamic pools of media resources. Application servers and other users of media servers (called media server clients) can define and create many different services based on how they configure and use those resources. Often, that configuration and the ways in which those resources interact will be changed dynamically over the course of a call, to reflect changes in the way that an application interacts with a user.

For example, a call may undergo an initial IVR dialog before being placed into a conference. Calls may be moved from a main conference to a sidebar conference and then back again. Individual calls may be directly bridged to create small n-way calls or simple sidebars. None of these change the SIP [1] dialog or RTP [10] session. Yet these do affect the media flow internal to the media server.

The Media Sessions Markup Language (MSML) is an XML [2] language used to change the flow of and services on media streams within a media server. It is used to invoke many different types of services on individual sessions, groups of sessions, and conferences. MSML allows the creation of conferences, bridging different sessions together, and bridging sessions into conferences.

MSML can be used to apply IVR operations and dialogs to sessions or conferences, and to modify the media flowing on a session. Dialogs may be specified in different languages depending on application requirements. VoiceXML [11] allows complete single-party application interfaces to be executed by a media server. Media Objects Markup Language (MOML) [12] can be used to specify individual user dialog or media control commands.

A network connection is established with the media server using SIP. Media received and transmitted on that connection will flow through different media resources on the media server depending on the requested service. Basic Network Media Services with SIP [14] defines conventions for associating a basic service with a SIP Request-URI. MSML allows services to be dynamically applied and changed by an application server during the lifetime of the SIP dialog.

MSML and MOML have been designed to work closely together: MOML addresses the control and manipulation of media processing operations (e.g., announcement, IVR, play and record, ASR/TTS, fax, video), while MSML addresses the relationships of media streams (e.g., simple and advanced conferencing). Together, MSML and MOML create a general-purpose media server control architecture. MSML can additionally be used to invoke other IVR languages such as VoiceXML.
2. Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [7].
3. Glossary

Media Server: a general-purpose platform for executing real-time media processing tasks. It may be a single physical device or a logical function within a physical device.

Media Server Client: an application residing on an external server which originates MSML requests to a media server.

Object: the generic term for a media server entity that terminates, originates, or processes media. This specification defines four classes of objects and specifies mechanisms to create them, join them together, and destroy them.

Participant Object: an object in a media server that sources original media in a call or that receives and terminates media in a call.

Intermediary Object: an object in a media server that acts on media within a call for the benefit of the participants.

Independent Object: an object that can exist on a media server independent of other objects.

Network Connection: a participant object class that represents the termination on a media server of one or more RTP [10] sessions (for example audio and video) associated with a call. Network connections are established and removed using a session establishment protocol such as SIP. An instance of a network connection is an independent object.

Operator: an intermediary object class that modifies or transforms a media stream. Examples of operators may be audio gain controls, video scaling, or voice masking. Specific types of operators are not defined within MSML. Operators may be defined in MOML [12] or other similar languages.

Dialog: an automated participant object class. Examples of dialogs may be announcement players, IVR interfaces, or voice recorders. Specific types of dialogs are not defined within MSML. Dialogs may be defined in VoiceXML [11], MOML, or other similar languages.

Conference: an intermediary object class that provides multimedia mixing and other advanced conferencing services. This specification currently considers conferences with audio and/or video media types, but is extensible to other media types. An instance of a conference is an independent object.

Identifier: a name that is used to refer to a specific instance of an
object on the media server. Identifiers are composed of one or more terms where each term identifies an object class and instance.

Media Stream: a single media flow between two objects. A media stream has a media type and may be unidirectional or bidirectional.

Media Stream Collection: a set of associated media streams between two objects that are treated as a single logical unit. An example of a media stream collection would be the audio and video streams between a network connection and a multimedia conference.
4. MSML SIP Usage

SIP is used to create and modify media sessions with a media server according to the procedures defined in RFC 3261 [1]. Often, SIP third party call control will be used to create sessions to a media server on behalf of end users. MSML is used to define and change the service which a user connected to a media server will receive. As such, MSML clients are expected to be application servers, which MUST have an authorized security relationship with the media server. MSML itself does not define authorization mechanisms.

MSML transactions are originated based upon events that occur in the application domain. These events may be independent from any media or user interaction. For example, an application may wish to play an announcement to a conference warning that its scheduled completion time is approaching. Applications themselves are structured in many different ways. Their structure and requirements contribute to their selection of protocols and languages. To accommodate differing application needs, MSML has been designed to be neutral to other languages and independent of the transport used to carry it.

Many alternatives exist for a transport mechanism for MSML. There may be one or many transport channels used to carry MSML based upon the requirements and structure of applications. SIP INVITE and INFO requests and responses have been chosen to carry MSML in this release of the specification. INFO requests allow asynchronous mid-call messages within SIP with few additional semantics. In addition, there are existing widely deployed implementations of that method, it aids in initial developments which are closely coupled with SIP session establishment, and it allows MSML to be directly associated with user dialogs when third party call control is used.

Although INFO is generally not considered to be a suitable general-purpose transport mechanism for messages within SIP, there have been proposals to make it more acceptable. MSML is expected to evolve to include other SIP usage and/or to work with other protocols or as a stand-alone protocol established through SIP, in future releases of this document.

MSML supports several models for client interaction. When clients use 3PCC to establish media sessions on behalf of end users, clients will have a SIP dialog for each media session. MSML MAY be sent on these dialogs. However the targets of MSML actions are not inferred from the session associated with the SIP dialog. The targets of MSML actions are always explicitly specified using identifiers as previously defined.

An application, after interacting with a user, may want to affect
multiple objects within a media server. For example, tones or messages are often played to a conference when connections are added or removed. A separate message may also be played to a participant as they are joined, or to moderators. Explicit identifiers not inferred from a transport mechanism allow these multiple actions to be easily grouped into a single transaction sent on any SIP dialog.

MSML also supports a model of dedicated control associations. This supports decoupled application architectures where a client can control media server services without also establishing all of the media sessions itself. Control associations are created using SIP but they do not have any associated media session. Although initially INFO messages will be sent on this SIP dialog, just as with dialogs associated with media sessions, it is expected that in the future, the SIP dialog will be used to establish a separate control session (defined in SDP [4]) that does not use SIP as the transport for MSML messages.

A media server using MSML also sends asynchronous events to a client using SIP INFO. Events are sent based on previous MSML requests and are sent within the SIP dialog on which the MSML request that caused the event to be generated was received. If the dialog no longer exists when the event is generated, the event is discarded.

Events may be generated during the execution of a dialog created by a "<dialogstart>" element. For example, dialogs defined in MOML can send events based on user input. VoiceXML dialogs on the other hand, generally interact with other servers outside of MSML using HTTP.

An event is also generated when the execution of a dialog terminates, either because of completion or failure. The exact information returned is dependent on the dialog language, the capabilities of the dialog execution environment, and what was specified by the dialog. Both MOML [12] and VoiceXML [11] allow information to be returned when they exit. These events may be sent in a SIP INFO or a SIP BYE. BYE is used when the dialog itself specifies that the connection should be disconnected such as through the use of <disconnect>.

Conferences may also generate events based upon their configuration. An example of this is the notification of the set of active speakers.
5. Language Structure

5.1. Package Scheme

The primary mechanism for extending MSML is the "package". A package is an integrated set of one or more XML schemas that define additional features and functions via new or extended use of elements and attributes. Each package is defined in a separate standards document, e.g., an Internet Draft or an RFC. All packages, extending the base MSML functionality, MUST include references to the MSML base set of schemas provided in the Internet drafts.

A schema in a package MUST extend MSML or MOML, i.e., may not mix MSML and MOML. A package MAY contain any combination of schemas extending MSML and schemas extending MOML.

A particular MSML or MOML script will include references to all the schemas defining the packages whose elements and attributes it makes use of. A particular script MUST reference either MSML base and optionally package(s) or MOML base and optionally package(s). See IANA Considerations section 15.

Each package MUST have its own namespace so that elements or attributes with the same name in different packages do not conflict. A script using a particular element or attribute MUST prefix the namespace name on that element or attribute’s name if it is defined in a package (as opposed to being defined in the base).

5.2. Profile Scheme

Not all devices and applications using MSML/MOML will need to support the entire MSML/MOML base schema. For example, a media processing device might support only audio announcements, only audio simple conferencing, or only multimedia IVR. It is high desirable to have a system for describing what portion of the MSML/MOML base a particular media processing device or application server supports.

Since packages are the scheme used for describing chunks of MSML/MOML functionality beyond the MSML/MOML base, one possibility might be to break the MSML/MOML base into a number of packages. MSML/MOML, however, is an XML-based language with sophisticated capabilities, and audio and video capabilities, for example, are intertwined in the schema. So it would not be technically feasible to break the MSML base and the MOML base into individual packages.

A much better solution is to create profiles of the MSML/MOML for the different uses. Each profile would identify a subset of the MOML/MOML base element and attributes, and each profile would be
accompanied by one or two corresponding schemas that are a subset of the MSML base and/or the MOML base. To use the examples above, there could be an audio announcements profile, an audio simple conferencing profile, and a multimedia IVR profile.

These profiles would be published separately from the MSML/MOML base documents, in one or more standards documents (e.g., Internet Drafts or RFCs) dedicated to profiles. Profiles would not be registered with IANA and any organization would additionally be free to create its own profile(s) if required.

5.3. Execution Flow

MSML assumes a model where there is a single control context within a media server for MSML processing. That context may have one or many SIP [1] dialogs associated with it. It is assumed that any SIP dialogs associated with the MSML control context have been authorized by mechanisms outside the scope of MSML.

A media server control context maintains information about the state of all media objects and media streams within a media server. It receives and processes all MSML requests from authorized SIP dialogs and receives all events generated internally by media objects and sends them on the appropriate SIP dialog. An MSML request is able to create new media objects and streams, and to modify or destroy any existing media objects and streams.

An MSML request may simply specify a single action for a media server to undertake. In this case, the document is very similar to a simple command request. Often, though, it may be more natural for a client to request multiple actions at one time, or the client would like several actions to be closely coordinated by the media server. Multiple MSML elements received in a single request MUST be processed sequentially in document order.

An example of the first scenario would be to create a conference and join it with an initial participant. An example of the second case would be to unjoin one or more participants from a main conference and join them to a sidebar conference. In the first scenario, network latencies may not be an issue, but it is simpler for the client to combine the requests. In the second case, the added network latency between separate requests could mean perceptible audio loss to the participant.

Each MSML request is processed as a single transaction. A media server MUST ensure that it has the necessary resources available to carry out the complete transaction before executing any elements of the request. If it does not have sufficient resources, it MUST
return a 520 response and MUST NOT execute the transaction.

The MSML request MUST be checked for well-formedness and validated against the schema prior to executing any elements. This allows XML errors to be reported immediately and minimizes failures within a transaction and the corresponding execution of only part of the transaction.

Each element is expected to execute immediately. Elements such as "<dialogstart>", which take time, are "forked" and executed in a separate thread. Once successfully forked, execution continues with the element following the dialog. As such, MSML does not provide mechanisms to sequence or coordinate other operations with dialog elements.

Processing within a transaction MUST stop if any errors occur. Elements that were executed prior to the error are not rolled back. It is the responsibility of the client to determine appropriate actions based upon the results indicated in the response. Most elements MAY contain an optional "mark" attribute. The value of that attribute from the last successfully executed element MUST be returned in an error response. Note that errors that occur during the execution of a dialog occur outside the context of an MSML transaction. These errors will be indicated in an asynchronous event.

Transaction results are returned as part of the SIP request response. The transaction results indicate the success or failure of the transaction. The result MUST also include identifiers for any objects created by a media server for which the client did not provide an instance name. Additionally, if the transaction fails, the reason for the failure MUST be returned, as well as an indication of how much of the transaction was executed before the failure occurred SHOULD be returned.

5.4. MSML Root Element

"<msml>" is the root element. When received by a media server, it defines the set of operations that form a single MSML transaction. Operations are requested by the contents of the element. Each operation MAY appear zero or more times as children of "<msml>". Specific operations are defined in sections 7 through 9.

The results of a request or the contents of events sent by a media server are also enclosed within the "<msml>" element. The results of the transaction are included as a body in the response to the SIP request that contained the transaction. This response will contain any identifiers that the media server assigned to newly created
objects. All messages that a media server generates are correlated to an object identifier. Objects and identifiers are discussed in section 6.

Attributes:

version: "1.1" Mandatory

5.5. Sending Events to a Media Server

5.5.1. <send>

Events are used to affect the behavior of different objects within a media server. The <send> element is used to send an event to the specified recipient.

attributes:

event: the name of an event. Mandatory.

target: an object identifier. When the identifier is for a dialog, it may optionally be appended with a slash "/" followed by the target to be included in a MOML <send>. Mandatory.

valuelist: a list of zero or more parameters that are included with the event.

mark: a token that can be used to identify execution progress in the case of errors. The value of the mark attribute from the last successfully executed MSML element is returned in an error response. Therefore the value of all mark attributes within an MSML document should be unique.

5.6. Transaction Results and Notifications

5.6.1. <result>

The <result> element is used to report the results of an MSML transaction. It is included as a body in the final response to the SIP request which initiated the transaction. An optional child element <description> may include text which expands on the meaning of error responses. Response codes are defined in section 10.

attributes:

response: a numeric code indicating the overall success or failure of the transaction, and in the case of failure, an indication of the reason. Mandatory.
mark: in the case of an error, the value of the mark attribute from the last successfully executed element that included the mark attribute.

In the case of failure, a description of the reason SHOULD be provided using the child element <description>.

Three other child elements allow the response to include identifiers for objects created by the request but which did not have instance names specified by the client. Those elements are <confid> and <dialogid>, for objects created through a <createconference> and <dialogstart> respectively.

5.6.2. <event>

The <event> element is used to notify an event to a media server client. Two types of events are defined by MSML: "msml.dialog.exit", "msml.conf.nomedia", and "msml.conf.asn". These correspond to the termination of an executing dialog, a conference being automatically deleted when the last participant has left, and the notification of the current set of active speakers for a conference, respectively. Events may also be generated by an executing dialog. In this case the event type is specified by the dialog.

attributes:

name: the type of event. Mandatory.

id: the identifier of the conference or dialog that generated the event or caused the event to be generated. Mandatory.

<event> has two children, <name> and <value>, which contain the name and value respectively of each namelist item associated with the event.
6. Media Server Object Model

Media servers are general-purpose platforms for executing real-time media processing tasks. These tasks range in complexity from simple ones such as serving announcements, to complex ones, such as speech interfaces, centralized multimedia conferencing, and sophisticated gaming applications.

Calls are established to a media server using SIP. Clients will often use SIP third party call control (3PCC) [13] to establish calls to a media server on behalf of end users. However MSML does not require that 3PCC be used; only that the client and the media server share a common identifier for the call and its associated RTP [10] sessions.

Objects represent entities which source, sink, or modify media streams. A media streams is a bidirectional or unidirectional media flow between objects on a media server. Media streams and operations on them are discussed in depth in section 7. The following subsections define the classes of objects that exist on a media server and the way these are identified in MSML.

6.1. Objects

A media object is an endpoint of one or more media streams. It may be a connection that terminates RTP sessions from the network or a resource that transforms or manipulates media. MSML defines four classes of media objects. Each class defines the basic properties of how object instances are used within a media server. However most classes require that the function of specific instances be defined by the client, using MSML or other languages such as VoiceXML, or the Media Objects Markup Language (MOML).

The following classes of media processing objects are defined. The class names are given in parentheses:

- network connection (conn)
- conference (conf)
- dialog (dialog)
- operator (oper)

Network connection is an abstraction for the media processing resources involved in terminating the RTP session(s) of a call. For audio services a connection instance presents a full-duplex audio stream interface within a media server. Multimedia connections have
multiple media streams of different media types, each corresponding to an RTP session. Network connections get instantiated through SIP [1].

A conference represents the media resources and state information required for a single logical mix of each media type in the conference (e.g. audio and video). MSML models multiple mixes/views of the same media type as separate conferences. Each conference has multiple inputs. Inputs may be divided into classes that allow an application to request different media treatment for different participants. For example, the video streams for some participants may be assigned to fixed regions of the screen while those for other participants may only be shown when they are speaking.

A conference has a single logical output per media type. For each participant, it consists of the audio conference mix, less any contributed audio of the participant, and the video mix shared by all conference participants. Some video conferences may have an optional ability to show the previous speaker to the current speaker when voice activated switching is used to select the video stream to display.

Conferences are instantiated using the "<createconference>" element. The content of the "<createconference>" element specifies the parameters of the audio and/or video mixes. Conferences are discussed in depth in section 8.

Dialogs are a class of objects that represent automated participants. They are similar to network connections from a media flow perspective and may have one or more media streams as the abstraction for their interface within a media server. Unlike connections however, dialogs are created and destroyed through MSML, and the media server itself implements the dialog participant.

The function that an instance of a dialog fulfills is defined by a client using a language such as VoiceXML or MOML. As such, "dialog" is a generic reference to the set of resources, both media and control, that are used to create either a simple action, such as an atomic play or record operation, or more complex application interface components, such as a VoiceXML interpreter. Dialogs are instantiated through the "<dialogstart>" element. Dialog operations are presented in section 9.

Operators are functions that are used to filter or transform a media stream. The function that an instance of an operator fulfills is defined as a property of the media stream. Operators may be unidirectional or bidirectional and have a media type. Unidirectional operators reflect simple atomic functions such as
automatic gain control, filtering tones from conferences, or applying specific gain values to a stream. Unidirectional operators have a single media input, which is connected to the media stream from one object, and a single media output, which is connected to the media stream of a different object.

Bidirectional operators have two media inputs and two media outputs. One media input and output is associated with the stream to one object and the other input and output is associated with a stream to a different object. Bidirectional objects may treat the media differently in each direction. For example, an operator could be defined which changed the media sent to a connection based upon recognized speech or DTMF received from the connection. Operators get instantiated when streams are created or modified using the elements "<join>" element and elements "<modifystream>" respectively.

The relationships between the different object classes is shown in the figure below.

A single, full-duplex instance of each object class is shown together with common relationships between them. An operator (such as gain) is shown between a connection and a conference and dialogs are shown participating both with an individual connection and with a conference. The figure is not meant to imply only one to one relationships. Conferences will often have hundreds of participants, and either connections or conferences may be interacting with more than one dialog. For example, one dialog may be recording a conference while other dialogs announce participants joining or leaving the conference.
6.2. Identifiers

Objects are referenced using identifiers that are composed of one or more terms. Each term specifies an object class and names a specific instance within that class. The object class and instance are separated by a colon ":" in an identifier term.

Identifiers are assigned to objects when they are first created. In general, either the MSML client or a media server may specify the instance name for an object. Objects for which a client does not assign an instance name will be assigned one by a media server. Media server assigned instance names are returned to the client as a complete object identifier in the response to the request that created the object.

It is meaningful for some classes of objects to exist independently on a media server. Network connections may be created through SIP at any time. MSML can then be used to associate their media with other objects as required to create services. Conferences may be created and have specific resources reserved waiting for participant connections.

Objects from these two classes, connections and conferences, are considered independent objects since they can exist on a standalone basis. Identifiers for independent objects consist of single term as defined above. For example, identifiers for a conference and connection could be "conf:abc" or "conn:1234" respectively. Clients which choose to assign instance names to independent objects must use globally unique instance names. One way to create globally unique names is to include the domain name of the client as part of the name.

Dialogs are only created to provide a service to independent objects. Dialogs may act as a participant in a conference or interact with a connection similar to a two participant call. Dialogs depend upon the existence of independent objects and this is reflected in the composition of their identifiers. Operators modify the media flow between other objects, such as application of gain between a connection and a conference. As operators are merely media transform primitives defined as properties of the media stream, they are not represented by identifiers.

Identifiers for dialogs are composed of a structured list of slash (/) separated terms. The left-most term of the identifier must specify a conference or connection. This serves as the root for the identifier. An example of an identifier for a dialog acting as a conference participant could be:
All objects except connections are created using MSML. Connections are created when media sessions get established through SIP. There are several options clients and media servers can use to establish a shared instance name for a connection and its media streams.

When media servers support multiple media types, the instance name SHOULD be a call identifier that can be used to identify the collection of RTP sessions associated with a call. When MSML is used in conjunction with SIP and third party call control, the call identifier MUST be the same as the local tag assigned by the media server to identify the SIP dialog. This will be the tag the media server adds to the "To" header in its response to an initial invite transaction. RFC 3261 requires the tag values to be globally unique.

An example of a connection identifier is: conn:74jgd63956ts.

With third party call control, the MSML client acts as a back to back user agent (B2BUA) to establish the media sessions. SIP dialogs are established between the client and the media server allowing the use of the media server local tag as a connection identifier. If this is not the case, a SIP event package MAY be used to allow a media server to notify new sessions to a client that has subscribed to this information.

Identifiers as described above allow every object in a media server to be uniquely addressed. They can also be used to refer to multiple objects. There are two ways in which this can currently be done:

- wildcards
- common instance names

An identifier can reference multiple objects when a wildcard is used as an instance name. MSML reserves the instance name comprised of a single asterisk ('*') to mean all objects that have the same identifier root and class. Instance names containing an asterisk cannot be created. Wildcards MUST only be used as the right most term of an identifier and MUST NOT be used as part of the root for dialog identifiers. Wildcards are only allowed where explicitly indicated below.

The following are examples of valid wildcards:
Examples of illegal wildcard usage are:

- conf:*/dialog:73849

Although identifiers share a common syntax, MSML elements restrict the class of objects which are valid in a given context. As an example, although it is valid to join two connections together, it is not valid to join two IVR dialogs.
7. Media Streams

Objects have at least one media input and output for each type of media that they support. Each object class defines the number of inputs and outputs of that class support. Media streams are created when objects are joined, either explicitly using "<join>" or implicitly when dialogs are created using "<startdialog>". Dialog creation has two stages, allocating and configuring the resources required for the dialog instance, and implicitly joining those resources to the dialog target during the dialog execution.

A join operation by default creates a bidirectional audio stream between two objects. Video and unidirectional streams may also be created. A media stream is created by connecting the output from one object to the input of another object and vice versa (assuming a bidirectional or full-duplex join).

Many objects may only support a single input for each type of media. Within this specification, only the conference object class supports an arbitrary number of inputs. When a stream is requested to be created to an object that already has a stream of the same type connected to its single input, the result of the request depends upon the type of the media stream.

Audio mixing is done by summing audio signals. Automatically mixing audio streams has common and straightforward applications. For example, the ability to bridge two streams allows for the easy creation of simple three-way calls or to bridge private announcements with a [whispered] conference mix for an individual participant. In the case of general conferences however, an MSML client SHOULD create an audio conference and then join participants to the conference. Conference mixers SHOULD subtract the audio of each participant from the mix so that they do not hear themselves.

A media server that receives a request that requires joining an audio stream to the single audio input of an object that already has an audio stream connected, SHOULD automatically bridge the new stream with the existing stream, creating a mix of the two audio streams. The maximum number of streams that may be bridged in this manner is implementation-specific. It is RECOMMENDED that a media server support bridging at least two streams. A media server that cannot bridge a new stream with any existing streams MUST fail the operation requesting the join.

Unlike audio mixing, there are many different ways that two video streams may be combined and presented. For example, they may be presented side by side in separate panes, picture in picture, or in a single pane which displays only a single stream at a time based on a
heuristic such as active speaker. Each of these options creates a very different presentation and require significantly different media resources.

A join operation does not describe how a new stream can be combined with an existing stream. Therefore automatic bridging of video is not supported. A media server MUST fail requests to join a new video stream to an object that only supports a single video input and already has a video stream connected to that input. For an object to have multiple video streams joined to it, the object itself must be capable in supporting multiple video streams. Conference objects can support multiple video streams and provide a way to specify the mixing presentation for the video streams.

A media server MUST NOT establish any streams unless the media server is able to create all the streams requested by an operation. Streams are only able to be created if both objects support a media type and at least one of the following conditions is true:

1. each object that is to receive media is not already receiving a stream of that type.
2. any object that is to receive media and is already receiving a stream of that type supports receiving an additional stream of that type. The only class of objects defined in this specification that directly support receiving multiple streams of the same type are conferences.
3. the media server is able to automatically bridge media streams for an object that is to receive media and that is already receiving a stream of the requested type. The only type of media defined in this specification that MAY be automatically bridged is audio.

The directionality of media streams associated with a connection are modeled independently from what SDP [4] allows for the corresponding RTP [10] sessions. Media servers MUST respect the SDP in what they actually transmit but MUST NOT allow the SDP to affect the directionality when joining streams internal to the media server.

The following three sub-sections describe the elements that establish, modify, and remove streams. These are followed by sub-sections describing stream properties and specialized methods for establishing a stream.

7.1. <join>

"<join>" is used to create one or more streams between two
independent objects. Streams may be audio or video and may be
bidirectional or unidirectional. A bidirectional stream is
implicitly composed of two unidirectional streams that can be
manipulated independently. The streams to be established are
specified by "<stream>" elements (section 7.4) as the content of
"<join>".

Without any content, "<join>" by default establishes a bidirectional
audio stream. When only a stream of a single type has previously
been created between two objects, or when only a unidirectional
stream exists, "<join>" can be used to add a stream of another media
type or make the stream bidirectional by including the necessary
"<stream>" elements. Bidirectional streams are made unidirectional
by using "<unjoin>" (section 7.3) to remove the unidirectional stream
for the direction that is no longer required.

In addition to defining the media type and direction of streams,
"<stream>" elements are also used to establish the properties of
streams, such as gain, voice masking, or tone clamping of audio
streams, or labels and other visual characteristics of video streams.
Properties are often defined asymmetrically for a single direction of
a stream. Creating a bidirectional stream requires two "<stream>"
elements within the "<join>", one for each direction, if one
direction is to have different properties from the other direction.

If a media server can provide services using both compressed or
uncompressed media, the MSML client may need to distinguish within
requests which format is to be used. When compressed streams are
created, both objects must use the same media format or an error
response (450) is generated.

attributes:

id1: an identifier of either a connection or conference. Wildcards
MUST NOT be used. Any other object class results in a 440 error.

id2: an identifier of either a connection or conference. Wildcards
MUST NOT be used. Any other object class results in a 440 error.

mark: a token which can be used to identify execution progress in the
case of errors. The value of the mark attribute from the last
successfully executed MSML element is returned in an error response.
Therefore the value of all mark attributes within an MSML document
SHOULD be unique.

For example, consider a call center coaching scenario where a
supervisor can listen to the conversation between an agent and a
customer, and provide hints to the agent, which are not heard by the
customer. One join establishes a stream between the agent and the customer and another join establishes a stream between the agent and the supervisor. A third join is used to establish a half-duplex stream from the customer to the supervisor. The media server automatically bridges the media streams from the customer and the supervisor for the agent, and from the customer and the agent for the supervisor.

Assuming the following connections, each with a single audio stream:

- conn:supervisor
- conn:agent
- conn:customer

The following would create the media flows previously described:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<msml version="1.1">
  <join id1="conn:supervisor" id2="conn:agent"/>
  <join id1="conn:agent" id2="conn:customer"/>
  <join id1="conn:supervisor" id2="conn:customer">
    <stream media="audio" dir="to-id1"/>
  </join>
</msml>
```

The following example shows joining a participant to a multimedia conference. It assumes that the conference has a video presentation region named "topright". The "display" attribute is explained in section 7.4.2.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<msml version="1.1">
  <join id1="conn:hd83t5hf7g3" id2="conf:example">
    <stream type="audio"/>
    <stream type="video" dir="from-id1" display="topright"/>
    <stream type="video" dir="to-id1"/>
  </join>
</msml>
```

7.2. <modifystream>

Media streams can have different properties such as the gain for an audio stream or a visual label for a video stream. These properties are specified as the content of "<stream>" elements (see section 7.4). "<modifystream>" is used to change the properties of a stream by including one or more "<stream>" elements that are to have their properties changed.
Stream properties MUST be set as specified by the element "<stream>" as a child element of "<modifystream>" element. Any properties not included in the "<stream>" element when modifying a stream MUST remain unchanged. Setting a property for only one direction of a bidirectional stream MUST NOT affect the other direction. The directionality of streams can be changed using issuing an "<unjoin>" followed by a "<join>". Any streams that exist between the two objects that are not included within "<modifystream>" MUST NOT be affected.

attributes:

id1: an identifier of either a conference or a connection. The instance name MUST NOT contain a wildcard if "id2" contains a wildcard. Mandatory.

id2: an identifier of either a conference or a connection. The instance name MUST NOT contain a wildcard if "id1" contains a wildcard. Mandatory.

mark: a token which can be used to identify execution progress in the case of errors. The value of the mark attribute from the last successfully executed MSML element is returned in an error response. Therefore the value of all mark attributes within an MSML document are RECOMMENDED to be unique.

7.3. <unjoin>

Unjoin removes one or more media streams between two objects. In the absence of any content in "<stream>" element, all media streams between the objects MUST be removed. Individual streams may be removed by specifying them using "<stream>" elements, while the unspecified streams MUST NOT be removed. A bidirectional stream is changed to a unidirectional stream by unjoining the direction that is no longer required, using the "<unjoin>" element. Operator elements MUST NOT be specified within "<stream>" elements when streams are being unjoined using the "<unjoin>" element. Any specified stream operators MUST be ignored.

"<unjoin>" and "<join>" may be used together to move a media stream, such as from a main conference to a sidebar conference.

attributes:

id1: an identifier of either a conference or a connection. The instance name MUST NOT contain a wildcard if "id2" contains a wildcard. Mandatory.
id2: an identifier of either a conference or a connection. The instance name MUST NOT contain a wildcard if "id1" contains a wildcard. Mandatory.

mark: a token which can be used to identify execution progress in the case of errors. The value of the mark attribute from the last successfully executed MSML element is returned in an error response. Therefore the value of all mark attributes within an MSML document SHOULD be unique.

The following removes a participant from a conference and plays a leave tone for the remaining participants in the conference.

```xml
<?xml version="1.0" encoding="UTF-8"?
<msml version="1.1">
  <unjoin id1="conn:jd73ht89sf489f" id2="conf:1"/>
  <dialogstart target="conf:1" type="application/moml+xml">
    <play>
      <audio uri="file://leave_tone.wav"/>
    </play>
  </dialogstart>
</msml>
```

7.4.  <stream> Elements

Individual streams are specified using the "<stream>" element. They MAY be included as a child element in any of the stream manipulation elements "<join>", "<modifystream>", or "<unjoin>".

The type of the stream is specified using a "media" attribute that uses values corresponding to the top-level MIME media types as defined in RFC 2046 [6]. This specification only addresses audio and video media. Other specifications may define procedures for additional types.

A bidirectional stream is identified when no direction attribute "dir" is present. A unidirectional stream is identified when a direction attribute is present. The "dir" attribute MUST have a value of "from-id1" or "to-id1" depending on the required direction. These values are relative to the identifier attributes of the parent element.

The compressed attribute is used to distinguish the compressed nature of the stream when necessary. It is implementation specific what is used when the attribute is not present. Joining compressed streams acts much like an RTP [10] relay.

The properties of the media streams are specified as the content of
"<stream>" elements when the element is used as a child of "<join>" or "<modifystream>". Stream elements MUST NOT have any content when they are used as a child of "<unjoin>" to identify specific streams to remove.

Some properties are defined within MSML as additional attributes or child elements of "<stream>" that are media type specific. Ones for audio streams and video streams are defined in the following two subsections. Operators, viewed as properties of the media stream, MAY be specified as child elements of the "<stream>" element.

attributes:

media: "audio" or "video". Mandatory

dir: "from-id1" or "to-id1".

compressed: "true" or "false". Specifies whether the stream uses compressed media. Default is implementation specific.

7.4.1. Audio Stream Properties

Audio mixes can be specified to only mix the N-loudest participants. However there may be some "preferred" participants that are always able to contribute. When audio streams are joined to a conference that uses N-loudest audio mixing, preferred streams need to be identified.

A preferred audio stream is identified using the "preferred" attribute. The "preferred" attribute MAY be used for an audio stream that is input to a conference and MUST NOT be used for other streams.

Additional attributes of the "<stream>" element for audio streams are:

preferred: a boolean value that defines whether the stream does not contend for N-loudest mixing. A value of "true" means that the stream MUST always be mixed while a value of "false" means that the stream MAY contend for mixing into a conference when N-loudest mixing is enabled. Default "false".

There are two elements that can be used to change the characteristics of an audio stream as defined below.

7.4.1.1. Gain

The "<gain>" element may be used to adjust the volume of an audio media stream. It may be set to a specific gain amount, to
automatically adjust the gain to a desired target level, or to mute the stream.

Attributes of the "<gain>" element are:

- **amt**: a specific gain to apply specified in dB or the string "mute" indicating that the stream should be muted. This attribute MUST NOT be used if "agc" is present.

- **agc**: boolean indicating whether automatic gain control is to be used. This attribute MUST NOT be used if "amt" is present.

- **tgtlvl**: the desired target level for AGC specified in dBm0. This attribute MUST be specified if "agc" is set to "true". This attribute MUST NOT be specified if "agc" is not present.

- **maxgain**: the maximum gain that AGC may apply. Maxgain is specified in dB. This attribute MUST be used if "agc" is present and MUST NOT be used when "agc" is not present.

### 7.4.1.2 Tone Removal

The "<clamp>" element is used to filter tones and/or audio-band dtmf from a media stream.

Attributes of the "<clamp>" element are:

- **dtmf**: boolean indicating whether DTMF tones should be removed.

- **tone**: boolean indicating whether other tones should be removed.

### 7.4.2 Video Stream Properties

Video mixes define a presentation that may have multiple regions, such as a quad-split. Each region displays the video from one or more participants. When video streams are joined to such a conference, the region that will display the video needs to be specified as part of the join operation.

The region that will display the video is specified using the "display" attribute. The "display" attribute MUST be used for a video stream that is input to a conference and MUST NOT be used for other streams. The value of the attribute MUST identify a "<region>" (section 8.5.2) or a "<selector>" (section 8.5.3) that is defined for the conference. A stream MUST NOT be directly joined to a region that is defined within a selector. Changing the value of the "display" attribute can be used to change where in a video presentation layout a video stream is displayed.
Additional attributes of the "<stream>" element for video streams are:

display: the identifier of a video layout region or selector that is to be used to display the video stream.

7.4.2.1. Visual Characteristics

Some regions of video conferences may display different streams automatically, such as when voice activated switching is used. Connections MAY also be joined directly without the use of video mixing. In these cases, the "<visual>" element may be used to define visual display properties for a stream.

The "<visual>" element MAY use any of the visual attributes defined for regions (see section 8.5.2). This allows the visual aspects of regions within a "<selector>" to be tailored to the selected video stream, or for streams that are directly joined to display a name or logo.

7.5. <monitor>

Monitor is a specialized unidirectional join that copies the media that is destined for a connection object. One example of the use for <monitor> may be quality monitoring within a conference. The media stream may be removed using the <unjoin> element described in section 7.3.

attributes:

id1: an identifier of the connection to be monitored. Any other object class results in a 440 error. Wildcards MUST NOT be used.

id2: an identifier of the object which is to receive the copy of the media destined to id1. id2 may be a connection or a conference. Any other object class results in a 440 error. Wildcards MUST NOT be used.

compressed: "true" or "false". Specifies whether the join should occur before or after compression. When "true", id2 must be a connection using the same media format as id1 or an error response (450) is generated. Default is "false.

mark: a token which can be used to identify execution progress in the case of errors. The value of the mark attribute from the last successfully executed MSML element is returned in an error response. Therefore the value of all mark attributes within an MSML document SHOULD be unique.
8. Conferences

A conference has a mixer for each type of media that the conference supports. Each mix has a corresponding description that defines how the media from participants contributes to that mix. A mixer has multiple inputs that are combined in a media specific way to create a single logical output.

The elements that describe the mix for each media type are called mixer description elements. They are:

- "<audiomix>" defines the parameters for mixing audio media.
- "<videolayout>" defines the composition of a video window.

These elements, defined in sections 8.4 and 8.5 respectively, are used as content of the "<createconference>" element to establish the initial properties of a conference. The elements are used within the "<modifyconference>" element to change the properties of a conference once it has been created, or within the "<destroyconference>" element to remove individual mixes from the conference.

Conferences may be terminated by an MSML client using the "<destroyconference>" element to remove the entire conference or by removing the last mixer(s) associated with the conference. Conferences can also be terminated automatically by a media server based on criteria specified when the conference is created. When the conference is deleted, any remaining participants will have their associated SIP dialogs left unchanged or deleted based on the value of the "term" attribute specified when the conference was created.

The following three sub-sections describe the elements that create, modify, and delete conferences. These are followed by sub-sections describing the properties for the mixers of each media type.

8.1. <createconference>

"<createconference>" is used to allocate and configure the media mixing resources for conferences. A description of the properties for each type of media mix required for the conference is defined within the content of the "<createconference>" element. Mixer descriptions are described in sections 8.4 and 8.5. When no mixer descriptions are specified, the default behavior MUST be equivalent to inclusion of a single "<audiomix>".

Clients can request that a media server automatically delete a conference when a specified condition occurs by using the "deletewhen" attribute. A value of "nomedia" indicates that the
conference MUST be deleted when no mixes remain into the conference. When this occurs, an "msml.conf.nomedia" event MUST be notified to the MSML client. A value of "nocontrol" indicates the conference MUST be deleted when the SIP [1] dialog that carries the "<createconference>" element is terminated. When this occurs, a media server MUST terminate all participant dialogs by sending a BYE for their associated SIP dialog. A value of "never" MUST leave the ability to delete a conference under the control of the MSML client.

attributes:

name: the instance name of the conference. If the attribute is not present, the media server MUST assign a globally unique name for the conference. If the attribute is present but the name is already in use, an error (432) will result and MSML document execution MUST stop. Events which the conference generates use this name as the value of their "id" attribute (see section 5.6.2).

deletewhen: defines whether a media server should automatically delete the conference. Possible values are "nomedia", "nocontrol", and "never". Default is "nomedia".

term: when true, the media server MUST send a BYE request on all SIP dialogs still associated with the conference when the conference is deleted. Setting term equal to false allows clients to start dialogs on connections once the conference has completed. Default true.

mark: a token which MAY be used to identify execution progress in the case of errors. The value of the mark attribute from the last successfully executed MSML element is returned in an error response. Therefore the value of all mark attributes within an MSML document should be unique.

An example of creating an audio conference is shown below. This conference allows at most two participants to contend to be heard and reports the set of active speakers no more frequently than every ten seconds.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<msml version="1.1">
  <createconference name="example">
    <audiomix>
      <n-loudest n="3"/>
      <asn ri="10s"/>
    </audiomix>
  </createconference>
</msml>
```
8.2.  <modifyconference>

All of the properties of an audio mix or the presentation of a video mix may be changed during the life of a conference using the "<modifyconference>" element. Changes to an audio mix are requested by including an "<audiomix>" element (see section 8.4) as a child of "<modifyconference>". This may also be used to add an audio mixer to the conference if none was previously allocated. Changes to a video presentation are requested by including a "<videolayout>" element (see section 8.5) as a child of "<modifyconference>". Similar to an audio mixer, this may be used to add a video mixer if none was previously allocated.

Mixers are removed by including a mixer description element within "<destroyconference/>".

Features and presentation aspects are enabled/added or modified by including the element(s) that define the feature or presentation aspect within a mixer description. The complete specification of the element must be included just as it would be included when the conference is created. The new definition completely replaces any previous definition that existed. Only things that are defined by elements included in the mixer descriptions are affected. Any existing configuration aspects of a conference, which are not specified within the "<modifyconference/>" element, MUST maintain their current state in the Media Server.

For example, if an MSML client wanted to change the minimum reporting interval for active speaker notification from that shown in the example in section 8.1 it would send the following to the media server:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<msml version="1.1">
  <modifyconference id1="conf:example">
    <audiomix>
      <asn ri="4">
      </audiomix>
  </modifyconference>
</msml>
```

This would also enable active speaker notification if it had not previously been enabled. The N-loudest mixing is unaffected.

Multiple elements MAY be included in the mixer descriptions similar to when conferences are created. For example, in a video conference, the video mix description ("<videolayout>") could specify that the layout of the video being displayed should change such that the
regions currently displaying participants get smaller and new region(s) are created to support additional participants. A media server MUST make all of the requested changes or none of the requested changes.

Additional examples of modifying conferences are presented in section 11.

attributes:

id: the identifier for a conference. Wildcards MUST NOT be used. Mandatory.

8.3. <destroyconference>

Destroy conference is used to delete mixers or to delete the entire conference and all state and shared resources. When a mixer is removed, all of the streams joined to that mixer are unjoined. When a conference is destroyed, SIP dialogs for any remaining participants MUST be maintained or removed based on the value of the "term" attribute when the conference was created.

When there is no element content, "<destroyconference/>" deletes the entire conference. Individual mixer(s) are removed by including a mixer description element identifying the mix(es) to be removed as content to "<destroyconference/>". "<audiomix/>" is used remove audio mixers and "<videolayout/>" is used remove video mixers. When one or more mixer descriptions are specified, then Media Server MUST only delete the specified mixer and MUST NOT affect any other existing mixers. When "<audiomix/>" or "<videolayout/>" are identified for individual removal, other feature aspects of the mix MUST NOT be included. If specified, the Media Server MUST ignore any such elements. When the last mixer is removed from a conference, a media server MUST remove all conference state, leaving or removing any remaining SIP dialogs as described above.

attributes:

id: the identifier for a conference. Mandatory.

mark: a token which can be used to identify execution progress in the case of errors. The value of the mark attribute from the last successfully executed MSML element is returned in an error response. Therefore the value of all "mark" attributes within an MSML document SHOULD be unique.
8.4. Audio Mix

The properties of the overall audio mix are specified using the "<audiomix>" element.

Attributes:

id: an optional identifier for the audio mix.

An example of the description for an audio mix is:

    <audiomix id="mix1">
      <asn ri="10s"/>
      <n-loudest n="3"/>
    </audiomix>

8.4.1. N-Loudest

The "<n-loudest>" element defines that participants contend to be included in the conference mix based upon their audio energy. When the element is not present, all participants are mixed.

Attributes:

n: the number of participants that will be included in the audio mix based upon having the greatest audio energy.

8.4.2. Active Speaker Notification

The "<asn>" element enables notification of active speakers. Active speakers MUST be notified using the "<event>" element (section 5.6.2) with an event name of "msml.conf.asn". The namelist of the event consists of the set of active speakers. The name of each item is the string "speaker" with a value of the connection identifier for the connection.

Attributes:

ri: the minimum reporting interval defines the minimum duration of time which must pass before changes to active speakers will be reported. A value of zero disables active speaker notification.
An example of an active speaker notification is:

```
<event name="msml.conf.asn" id="conf:example">
  <name>speaker</name>
  <value>conn:hd93tg5hdf</value>
  <name>speaker</name>
  <value>conn:w8cn59vei7</value>
  <name>speaker</name>
  <value>conn:p78fnh6sek47fg</value>
</event>
```

8.5. Video Layout

A video layout is specified using the "<videolayout>" element. It is used as a container to hold elements that describe all of the properties of a video mix. The parameters of the window that displays the video mix are defined by the "<root>" element. When the video mix is composed of multiple panes, the location and characteristics of the panes are defined by one or more "<region>" elements. A "<region>" element is not required when only a single video stream is displayed at one time and none of the visual attributes of regions are required.

Some regions may be used to display a video stream based on a selection criteria rather than having a video stream of a single participant continuously presented in the region. One such an example is a distance learning lecture where the instructor sees each of the students periodically displayed in a region. When a region is used to display one of a number of streams, it is placed as a child of a "<selector>" element.

Attributes:

- **type**: specifies the language used to define the layout. Layouts defined using MSML MUST use the value "text/msml-basic-layout". This is the same convention as defined for the layout module from the W3C SMIL 2.0 specification [15]. The default when omitted is "text/msml-basic-layout".

- **id**: an optional identifier for the video layout.

8.5.1. Root Layout

The "<root>" element describes the root window or virtual screen in which the conference video mix will be displayed. Simple conferences can display participant video directly within the root window but more complex conferences will use regions for this purpose. Areas of the window which are not used to display video will show the root
window background.

All video presentations require a root window. It MUST be present when a video mix is created and it cannot be deleted, however its attributes MAY be changed using the "<modifyconference>" element.

Attributes:

- **size**: the size of the root window specified as one of the five standard common intermediate formats (e.g. CIF, QCIF, etc.).

- **bgcolor**: the color for the root window background defined using the values for the "background-color" property of the CSS2 specification [8].

- **backgroundimage**: the URI for an image to be displayed as the root window background. Transparent portions of the image allow the background color to show through.

8.5.2. Regions

"<region>" elements define video panes that are used to display participant video streams. Regions are rendered on top of the root window.

The size of a region is specified relative to the size of the root window using the "relativesize" attribute. Relative sizes are expressed as fractions (e.g. 1/4, 1/3) that preserve the aspect ratio of the original video stream while allowing for efficient scaling implementations.

Regions are located on the root window based on the value of the position attributes "top" and "left". These attributes define the position of the top left corner of the region as an offset from the top left corner of the root window. Their values may be expressed either as a number of pixels or as a percent of the vertical or horizontal dimension of the root window. Percent values are appended with a percent ('%') character. Percent values of "33%" and "67%" should be interpreted as "1/3" and "2/3" to allow easy alignment of regions whose size is expressed relative to the size of the root window.
An example of a video layout with six regions is:

```
+-------+---+
|       | 2 |
|   1   +---+
|       | 3 |
+---+---+---+
| 6 | 5 | 4 |
+---+---+---+
```

```xml
<videolayout type="text/msml-basic-layout">
  <root size="CIF"/>
  <region id="1" left="0" top="0" size="2/3"/>
  <region id="2" left="67%" top="0" size="1/3"/>
  <region id="3" left="67%" top="33%" size="1/3"/>
  <region id="4" left="67%" top="67%" size="1/3"/>
  <region id="5" left="33%" top="67%" size="1/3"/>
  <region id="6" left="0" top="67%" size="1/3"/>
</videolayout>
```

The area of the root window covered by a region is a function of the region’s position and its size. When areas of different regions overlap, they are layered in order of their "priority" attribute. The region with the highest value for the "priority" attribute is below all other regions and will be hidden by overlapping regions. The region with the lowest non-zero value for the "priority" attribute is on top of all other regions and will not be hidden by overlapping regions. The priority attribute may be assigned values between 0 and 1. A value of zero disables the region, freeing any resources associated with the region, and unjoining any video stream displayed in the region.

Regions that do not specify a priority will be assigned a priority by a media server when a conference is created. The first region within the "<videolayout>" element that does not specify a priority will be assigned a priority of one, the second a priority of two, etc. In this way, all regions that do not explicitly specify a priority will be underneath all regions that do specify a priority. As well, within those regions that do not specify a priority, they will be layered from top to bottom, in the order they appear within the "<videolayout>" element.
For example, if a layout was specified as follows:

```xml
<videolayout>
  <root size="CIF"/>
  <region id="a" ... priority=".3" .../>
  <region id="b" ... />
  <region id="c" ... priority=".2" ...>
    <region id="d" ... />
  </region>
</videolayout>
```

Then the regions would be layered, from top to bottom, c,a,b,d.

Portions of regions that extend beyond the root window will be cropped. For example, a layout specified as:

```xml
<videolayout>
  <root size="CIF"/>
  <region id="foo" left="50%" top="50%" size="2/3"/>
</videolayout>
```

would appear similar to:

```
+-----------+
| root |
| background |
| +--------+
| |   // |
| | foo  // |
| +--------+//

```

Visual attributes are used to define aspects of the visual appearance of individual regions. A border may be defined together with a title and/or logo. Text and logos are displayed as images on top of the region's video, below all regions with a lower priority. The visual attributes are "title", "titletextcolor", "titlebackgroundcolor", "bordercolor", "borderwidth", and "logo".

Visual attributes can also be defined for individual streams (section 7.4.2). When visual attributes are specified as part of both a region and a stream, those associated with the stream MUST take precedence. This allows streams that are chosen for display automatically (see 8.5.3 to have proper text and logos displayed. The region visual attributes are displayed when no stream is associated with the region.

Two other attributes associated with a region, "blank" and "freeze", define the state of the video displayed in the region. When the

---

Saleem & Sharratt          Expires April 24, 2006        [Page 38]
blank or freeze attribute is assigned the value "true", then the Media Server MUST display the region either as a blank region, or the video image frozen at the last received frame.

Open Issue: these attributes are specified for a region and not allowed for streams because that appears to be the common use case. Applying them to streams would allow only that stream to be affected within a selector while other streams continue to display normally. Except for personal mixing scenarios, the same effect can be achieved by having the participant mute their own transmission to the media server.

Attributes associated with each region are:

id: a name that can be used to refer to the region.

left: the position of the region from the left side of the root window.

top: the position of the region from the top of the root window.

relativesize: the size of the region expressed as a fraction of the root window size.

priority: a number between 0 and 1 that is used to define the precedence when rendering overlapping regions. A value of zero disables the region.

title: text to be displayed as the title for the region

titletextcolor: the color of the text

titlebackgroundcolor: the color of the text background

bordercolor: the color of the region border

borderwidth: the width of the region border

logo: the URI of an image file to be displayed

freeze: a boolean value, with a default of false, that defines whether the video image should be frozen at the currently displayed frame

blank: a boolean value, with a default of false, that defines whether the region should display black instead of the associated video stream
8.5.3. Stream Selection

It is often desired that one of several video streams be automatically selected to be displayed. The "<selector>" element is used to define the selection criteria and its associated parameters. The selection algorithm is specified by the "method" attribute. Currently defined selection methods allow for voice activated switching and to iterate sequentially through the set of associated video streams.

The regions that will display the selected video stream are placed as child elements of the "<selector>" element. Including regions within a "<selector>" element does not affect their layout with respect to regions not subject to the selection. For simple video conferences that display the video directly in the root window, the "<root>" element can be placed as a child of "<selector>". Region elements MUST NOT be used in this case.

For example, below is a common video layout that allows the video stream from the currently active speaker to be displayed in the large region ("1") at the top left of the layout while the streams from five other participants are displayed in regions located at the layout periphery.

```
+-------+---+
|       | 2 |
|   1   +---+
|       | 3 |
+-------+---+
| 6 | 5 | 4 |
+---+---+---+
```

```xml
<videolayout type="text/msml-basic-layout">
  <root size="CIF"/>
  <selector id="switch" method="vas">
    <region id="1" left="0" top="0" size="2/3"/>
  </selector>
  <region id="2" left="67%" top="0" size="1/3"/>
  <region id="3" left="67%" top="33%" size="1/3"/>
  <region id="4" left="67%" top="67%" size="1/3"/>
  <region id="5" left="33%" top="67%" size="1/3"/>
  <region id="6" left="0" top="67%" size="1/3"/>
</videolayout>
```

All selector methods must be defined so that they work if only a single region is a child of the selector. Selector methods that support more than one child region MUST specify how the method works across multiple regions. Media server implementations MAY support
only a single region for methods that are defined to allow multiple regions.

The selector or region for a participant’s video is defined using the "display" attribute of "<stream>" during a join operation. Specifying a selector allows the stream to be displayed according to the criteria defined by the selector method. Specifying a region supports continuous presence display of participants. Some streams may be joined with both a selector and a region. In this case, the value of "<blankothers>" attribute defines whether the streams associated with a continuous presence region should be blanked when the stream is selected for display in one of the selector regions.

Attributes common to all selector methods are:

- id: a name that can be used to refer to the selector.
- method: the name of the method used to select the video stream.
- status: specifies whether the selector is "active" or "disabled".
- blankothers: when "true", video streams that are also displayed in continuous presence regions will have the continuous presence regions blanked when the stream is displayed in a selection region.

8.5.3.1. Voice Activated Switching

Voice activated switching (VAS) is used to display the video stream that correlates with the participant who is currently speaking. It is specified using a selector method value of "vas".

If the video stream associated with the active speaker is not currently displayed in a selection region, then it replaces the video in the region that is displaying the video of the speaker that was least recently active. If the video of the active speaker is currently displayed in a selection region, then there is no change to any region. When VAS is applied to a single region, this has the effect that the current speaker is displayed in that region.

Attributes associated with voice activated switching are:

- si: switching interval is the minimum period of time that must elapse before allowing the video to switch to the active speaker.
- speakerssees: defines whether the active speaker sees the "current" speaker (themselves) or the "previous" speaker.
8.6. Reserving Conference Resources

Conference resources may be reserved by including the "<reserve>" element as a child of "<createconference>". "<reserve>" allows the specification of a set of resources which a media server will reserve for the conference. Any requests for resources beyond those that have been reserved should be honored on a best-effort basis by a media server.

attributes:

required: boolean that specifies whether <createconference> should fail if the requested resources are not available. When set to false, the conference will be created, with no reserved resources, if the complete reservation cannot be honored. Default true.

The resources to be reserved are defined using "<resource>". The contents of these elements describe a resource that is to be reserved. Descriptions are implementation-dependent. Media servers that support MOML may use the elements from that language as the basis for resource descriptions. Each resource element may use the attribute "n" to define the quantity of the resource to reserve.

For example, the following creates a conference and reserves two types of resources. One resource element may represent resources that are shared by all participants of the conference with the other may represent resources that are reserved for each of the expected participants.

```xml
<createconference>
  <reserve>
    <resource n="20"> <!-- description of resources used by each participant -->
  </individual>
  <resource n="2"> <!-- description of the shared conference resources -->
    <shared>
  </shared>
  </reserve>
</createconference>
```
9. Dialogs

Dialogs are used for interaction with a user. A dialog may consist of a simple atomic command such as playing an announcement, or it may be an entire sequence of interactions. Dialogs may be speech or IVR dialogs with human participants, or fax dialogs with a machine. A media server MUST support MOML [12] to allow command driven and fax interactions, and SHOULD support VoiceXML to allow execution of complex user interfaces. Other dialog languages MAY also be supported.

The control resources associated with dialogs are separate from the MSML thread of execution. When a dialog is started, MSML allocates the dialog control resources, and if successful, starts those resources executing. MSML execution then continues without waiting for the dialog to complete.

Media streams are created between the dialog target and other internal media server resources as part of dialog execution. Stream creation is subject to the requirements defined in section 7.

9.1. <dialogstart>

"<dialogstart>" is used to instantiate media dialog on connections or conferences. The dialog is specified either inline or by a URI [3]. The dialog description MUST NOT be inline if the src attribute is present.

The originator of the dialog is notified using a "msml.dialog.exit" event when the dialog completes. Any results returned by the dialog when it exits are sent as a namelist to the event.

The "msml.dialog.exit" event is also used when dialogs fail due to errors encountered fetching external documents or errors that occur within the dialog execution thread. In this case, a namelist containing the items "dialog.exit.status" and "dialog.exit.description" is returned with the event to inform the client of the failure and the failure reason. The values of these items are defined in section 9. Information from the failed dialog may be returned as additional namelist items.

attributes:

target: an identifier of a connection or a conference which will interact with the dialog. The identifier must not contain wildcards. Mandatory.

src: the URL of the dialog description. MUST NOT be used if the
dialog description is inline. Otherwise an error (422) will result and MSML document execution will stop.

type: a MIME type which identifies the type of language used to describe the dialog. application/moml+xml and application/vxml+xml are used to identify MOML and VoiceXML [11] respectively.

name: an instance name for the dialog. If the attribute is not present, the media server will assign an identifier to the dialog. If the attribute is present but the name is already associated with the target, an error (431) will result and MSML document execution will stop. Any results that a dialog generates will be correlated to its identifier.

mark: a token which can be used to identify execution progress in the case of errors. The value of the mark attribute from the last successfully executed MSML element is returned in an error response. Therefore the value of all "mark" attributes within an MSML document should be unique.

The following example starts a VoiceXML dialog on a connection.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<msml version="1.1">
  <dialogstart target="conn:abcd1234" type="application/vxml+xml" name="sample"
    src="http://server.example.com/scripts/foo.vxml"/>
</msml>
```

If this dialog failed once its execution thread had begun, for example the fetch of the VoiceXML document failed, an example of the event which would be returned would be:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<event name="msml.dialog.exit"
  id="conn:abcd1234/dialog:sample">
  <name>dialog.exit.status</name>
  <value>423</value>
  <name>dialog.exit.description</name>
  <value>External document fetch error</value>
</event>
```

9.2.  <dialogend>

Dialog end is used to terminate a dialog created through <dialogstart> before it completes of its own accord. The operation of <dialogend> depends on the dialog language being used by the
executing context. When that context is VoiceXML, a "connection.disconnected" event will be thrown to the VoiceXML application. When that context is MOML, a "terminate" event will be sent to the MOML context.

<dialogend> allows the executing dialog the opportunity to gracefully complete before generating a "msml.dialog.exit" event. Dialog results may be returned and will be contained as a namelist to that event.

attributes:

id: the identifier of a dialog. Mandatory.

mark: a token which can be used to identify execution progress in the case of errors. The value of the mark attribute from the last successfully executed MSML element is returned in an error response. Therefore the value of all "mark" attributes within an MSML document should be unique.

For example, if the dialog from the previous example was still executing, the following would terminate the dialog and generate a "msml.dialog.exit" event.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<msml version="1.1">
  <dialogend id="conn:abcd1234/dialog:sample"/>
</msml>
```
10. Response Codes

The response codes defined in this section are returned as the value of the response attribute to the <result> element. Some values may also be returned as part of a namelist to an "msml.dialog.exit" event generated when an executing dialog fails.

Informational (1xx)

Reserved for future use

Success

200 Ok

Client Error (4xx)

400 Bad Request
401 Unknown Element
402 Unsupported Element
403 Missing mandatory element content
404 Forbidden element content
405 Invalid element content
406 Unknown attribute
407 Attribute not supported
408 Missing mandatory attribute
409 Forbidden attribute is present
410 Invalid attribute value
420 Unsupported media description language
421 Unknown media description language
422 Ambiguous request (both URI and inline description)
423 External document fetch error
424 Syntax error in foreign language
425 Semantic error in foreign language
426 Unknown error executing foreign language
430 Object does not exist
431 Object instance name already used
432 Conference name already in use
433 reserved
434 External document fetch error
440 Cannot join objects of the specified class
441 Objects have incompatible media types
442 reserved
443 reserved
444 Number of media inputs exceeded
450 Objects have incompatible media formats
451 Incompatible media stream format

Server Error (5xx)
500 Internal media server error
510 Not in service
511 Service Unavailable
520 No resource to fulfill request
521 Internal limit exceeded
11. Examples

These examples focus on the MSML used by an Application Server (AS) to control services on a Media Server (MS). They show the relationship between SIP signaling to establish media sessions and MSML service control commands. For brevity, only the content of MSML messages is shown. The examples assume that the AS and MS use the IPv4 address and UDP port number of the audio stream (on the MS) to identify the MSML connection.

11.1. Establishing a Dial-In Conference

```
<table>
<thead>
<tr>
<th>UA</th>
<th>Application Server</th>
<th>Media Server</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INVITE F1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200 F2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ACK F3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>createconference&gt; F4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200 F5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INVITE (SDP UA) F6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>--------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INVITE (SDP UA) F7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>--------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200 (SDP MS) F8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ACK F9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>--------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200 (SDP MS) F10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ACK F11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>--------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;startdialog&gt; F12</td>
<td></td>
</tr>
<tr>
<td></td>
<td>--------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200 F13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HTTP interactions F14</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;-------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;event&gt;(dialog.exit) F15</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;-------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;join&gt; F16</td>
<td></td>
</tr>
</tbody>
</table>
```

Steps 1-3: establish an MSML control channel for the conference. Alternatively, a control channel could already have been established which was used for all AS/MS interactions. A control channel per conference is only one possible model. Currently MSML uses SIP INFO requests and responses on this SIP dialog. There is a proposal to use this message exchange to establish a TCP channel for MSML similar to the approach used for MRCPv2. This approach would require that a request identifier be added to the <msml> element to correlate requests and responses. This currently relies on the SIP INFO request and response for this property. MSML messages are shown without specifying the transport in this example but it assumes a request/response correlation based on transport messages.

Step 4: create a conference that will mix the loudest two speakers and report those speakers to the application server every ten seconds. The media server will automatically terminate remaining media sessions and delete the conference and associated resources and when the control channel is terminated.

```
<msml version="1.1">
  <createconference name="exampleConf" deletewhen="nocontrol"

  <audiomix>
    <n-loudest n="3"/>
    <asn ri="10s"/>
  </audiomix>

  </createconference>
</msml>
```
Step 5: conference created successfully

```xml
<msml version="1.1">
  <result response="200"/>
</msml>
```

Steps 6-11: standard 3PCC establishment of a user initiated media session to a media server. This is the equivalent of a dial-in conference participant. The "To:" header returned by the MS in the 200 response of Step F8 was:

```
To: <sip:msml@ms.example.com>;tag=jd87dfg4h
```

Step 12: request an initial dialog with the participant to prompt for their name, desired conference, etc. The dialog completes by informing the participant they are joining the conference. If this was not the first participant, the dialog could also announce the other participants.

```xml
<msml version="1.1">
  <dialogstart target="conn:jd87dfg4h"
    type="application/vxml+xml"
    src="http://server.example.com/scripts/initial.vxml"/>
</msml>
```

Step 13: dialog started successfully. The dialog identifier is returned.

```xml
<msml version="1.1">
  <result response="200"/>
  <dialogid>conn:jd87dfg4h/dialog:12v5tq9</dialogid>
</msml>
```

Step 14: sequence of HTTP VoiceXML dialog interactions.

Step 15: the VoiceXML browser exits (but does not disconnect). If a namelist had been specified within the VoiceXML <exit> element, it would have been included in the <event> sent to the AS.

```xml
<msml version="1.1">
  <event name="msml.dialog.exit"
    id="conn:jd87dfg4h/dialog:12v5tq9"/>
</msml>
```
Step 16: join the participant to the conference and have the volume of their contributing audio automatically adjusted to a target level of -20 dBm0.

<msml version="1.1">
  <join id1="conn:jd87dfg4h" id2="conf:exampleConf">
    <stream media="audio" dir="from-id1">
      <gain agc="true" tgtlvl="-20"/>
    </stream>
    <stream media="audio" dir="to-id1"/>
  </msml>

Step 17: successfully joined to conference

<msml version="1.1">
  <result response="200"/>
</msml>

Steps 6 through 17 are repeated for the second participant.

Step 18: play a join tone or message announcing the new participant to the conference.

<msml version="1.1">
  <dialogstart target="conf:exampleConf" type="application/vnd.vxml+xml" src="http://server.example.com/scripts/joinmsg.vxml"/>
</msml>

Step 19: dialog started successfully. The dialog identifier is returned.

<msml version="1.1">
  <result response="200"/>
  <dialogid>conf:ExampleConf/dialog:j6fs8745</dialogid>
</msml>

Step 20: HTTP VoiceXML dialog interaction(s).

Step 21: the VoiceXML browser exits.

<msml version="1.1">
  <event name="msml.dialog.exit" id="conf:ExampleConf/dialog:j6fs8745"/>
</msml>

Steps 6 through 21 are repeated for the third and subsequent participants.
11.2. Example of a Sidebar Audio Conference

This example assumes that a conference has already been established as in the previous example. It creates a sidebar conference that hears the main conference as a whisper. Three participants are moved to the sidebar. After some period of time, the sidebar participants are returned to the main conference and the sidebar is deleted.

Step 1: the sidebar conference is created. It is joined half-duplex to the main conference and a manual gain object is inserted in the media stream. Three participants are then moved from the main conference to the sidebar. Although not shown, an AS could include the "mark" attribute in each element to allow recovery in the event of a mid-transaction error.

<msml version="1.1">
  <createconference name="sidebarConf" deletewhen="nomedia">
    <audiomix/>
  </createconference>
  <join id1="conf:sidebarConf" id2="conf:exampleConf">
    <stream media="audio" dir="to-id1">
      <gain amt="-20"/>
    </stream>
  </join>
  <unjoin id1="conn:gs5s4-1" id2="conf:exampleConf"/>
  <join id1="conn:gs5s4-1" id2="conf:sidebarConf"/>
  <unjoin id1="conn:hd764gr9-2" id2="conf:exampleConf"/>
  <join id1="conn:hd764gr9-2" id2="conf:sidebarConf"/>
  <unjoin id1="conn:h37frdvgs65-3" id2="conf:exampleConf"/>
  <join id1="conn:h37frdvgs65-3" id2="conf:sidebarConf"/>
</msml>

Step 2: sidebar conference created successfully and participants joined.

<msml version="1.1">
  <result response="200"/>
</msml>
Step 3: once the sidebar conference has completed, the participants are rejoined to the main conference. The sidebar is destroyed automatically by the MS when the last media stream is removed as specified when the sidebar conference was created.

```xml
<msml version="1.1">
  <unjoin id1="conn:gs5s4-1" id2="conf:sidebarConf"/>
  <join id1="conn:gs5s4-1" id2="conf:exampleConf"/>
  <unjoin id1="conn:hd764gr9-2" id2="conf:sidebarConf"/>
  <join id1="conn:hd764gr9-2" id2="conf:exampleConf"/>
  <unjoin id1="conn:h37frdvgs65-3" id2="conf:sidebarConf"/>
  <join id1="conn:h37frdvgs65-3" id2="conf:exampleConf"/>
</msml>
```

Step 4: participants successfully moved to main conference and sidebar destroyed.

```xml
<msml version="1.1">
  <result response="200"/>
</msml>
```

11.3. Example of Removing Conference

This example assumes a conference created similar to the first example where there is an MSML control channel specific to the conference and the conference has been configured to be deleted when that channel is removed (using SIP).

Steps 1-2: the AS signals BYE for the dialog used to establish the conference control channel.

Steps 3-6: the MS initiates terminating the media sessions for each participant remaining in the conference.

The MS deletes the conference and removes all resources when the last participant has been removed.
11.4. Example of Modifying a Video Layout

Assume that a conference named "example" is created using the following mixer descriptions.

+---+---+
| 1 | 2 |
+---+---+
| 3 | 4 |
+---+---+

```xml
<createconference name="quad-split">
  <audiomix>
    <n-loudest n="3"/>
    <asn ri="10s"/>
  </audiomix>
  <videolayout>
    <root size="CIF" background="white"/>
    <selector id="default" method="vas" si="500ms">
      <region id="1" left="0" top="0" size="1/4"/>
    </selector>
    <region id="2" left="50%" top="0" size="1/4"/>
    <region id="3" left="0%" top="50%" size="1/4"/>
    <region id="4" left="50%" top="50%" size="1/4"/>
  </videolayout>
</createconference>
```

The following would change the size of the video window to QCIF and the background color to the default "black".

```xml
<modifyconference id1="conf:example">
  <videolayout>
    <root size="4CIF"/>
  </videolayout>
</modifyconference>
```

The relative location of the regions does not change. However the sizes of the regions do change because they are relative to the size of the root window. The result is a layout that looks identical but half the size.

The following would freeze the video displayed in region "2" without affecting any other attributes of that region.

```xml
<modifyconference id1="conf:example">
  <videolayout>
    <region id="2" left="50%" top="0" size="1/4" freeze="true"/>
  </videolayout>
</modifyconference>
```
</modifyconference>
12. Change Summary

The following is a summary of changes between the -06 version of the draft and the -05 version:

- This version is derived from earlier versions co-authored by Tim Melanchuk and Garland Sharratt.
- Restated the description of operators to indicate that they are not MOML objects. Rather, operators are media transform functions operating on or defined as properties of media streams. Operators are defined as primitive MSML elements, such as gain.
- Removed references to "<insert>" element, which is not required since operators defined as MOML objects are no longer inserted.
- Clarifications added in several places indicating expected behavior when elements or attributes are omitted.
- Changed MSML version number from 1.0 to 1.1

The following is a summary of changes between the -05 version of the draft and the -04 version:

- none

The following are the primary changes between the -03 version of the draft and the -02 version:

- added framework for multimedia streams and defined video streams.
- added descriptions for audio mixes and video presentation layouts. The audio mix description replaces the "<createconference>" attributes "n", "asn" and, "ri".
- added the "<stream>" element to define a stream and several child elements that define properties of a stream. Made the "<insert>" element a child of "<stream>".
- added the elements "<modifystream>" and "<modifyconference>" to modify streams and conferences respectively.
- moved the "term" attribute from "<destroyconference>" to "<createconference>" so that it can affect the behavior when conferences are automatically deleted.
o deprecated the "<remove>" and "<cjoin>" elements. Removing operators is now accomplished as part of "<modifystream>". Compressed join is no longer necessary because "<stream>" elements allow compressed media to be identified and a compressed join can be accomplished using the standard "<join>" element.

The following are the primary changes between the -02 version of the draft and the -01 version:

o added the specialized join operations <cjoin> and <monitor>

o added "deletewhen" attribute to <createconference> to allow a media server to automatically delete conferences when the specified condition occurs

o clarified that <join> is used to change the duplexity of a media stream

The following are the primary changes between the -01 version of the draft and the -00 version.

o added a glossary

o rewrote the description of objects to precisely distinguish between classes and instances. All classes are now defined in MSML. The "oper" class replaces "application defined classes".

o rewrote the description of identifiers. All terms must use "class:instance" where the instance may be assigned by the client or media server. "/" replaces ";" as the term separator for identifiers.

o clarified the definition of connection identifiers and require that "conn" be the class for all forms of the identifier.

o '*' wildcard allowed for an instance name in limited situations.

o alias only names a single connection.

o clarified SIP usage and transport neutrality. All actions use mandatory explicit identifiers rather than inferring targets from a SIP dialog.

o changed the attribute name from "id" to "name" for client assigned instance names.

o fixed <destroy> so that MOML target is appended to the MSML target rather than the MSML event.
o changed xml+moml to moml+xml and xml+vxml to vxml+xml.
o changed "namelist" to "valuelist" in send.
o removed explicit "lhs" / "rhs" labeling of full duplex objects.
o added specification of result codes and when they are returned.
13. XML Schemas

The MSML schema uses one core schema which includes two other schemas; one defines the MSML datatypes, the other is for MOML which is optionally used for dialogs.

The core schema is:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified"
           attributeFormDefault="unqualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="moml.xsd"/>
  <xs:include schemaLocation="msml-datatypes.xsd"/>
  <xs:element name="msml">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:group ref="msmlRequest" maxOccurs="unbounded"/>
        <xs:element name="event">
          <xs:complexType>
            <xs:choice maxOccurs="unbounded">
              <xs:sequence>
                <xs:element name="name" type="msmlEventNameValue.datatype"/>
                <xs:element name="value">
                  <xs:simpleType>
                    <xs:restriction base="xs:string">
                      <xs:pattern value="[a-zA-Z0-9.-]+"/>
                    </xs:restriction>
                  </xs:simpleType>
                </xs:element>
              </xs:sequence>
              <xs:attribute name="name" type="msmlEventName.datatype" use="required"/>
              <xs:attribute name="id" type="msmlEventSource.datatype" use="required"/>
            </xs:choice>
          </xs:complexType>
        </xs:element>
        <xs:element name="result">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="description" minOccurs="0">
                <xs:simpleType>
                  <xs:restriction base="xs:string">
                    <xs:pattern value="[a-zA-Z0-9.-]+"/>
                  </xs:restriction>
                </xs:simpleType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
<xs:choice minOccurs="0" maxOccurs="unbounded">
  <xs:element name="confid" type="confID.datatype"/>
  <xs:element name="dialogid" type="dialogID.datatype"/>
</xs:choice>

<xs:attribute name="response">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="\d{3}"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:attribute name="mark" type="mark.datatype"/>
</xs:complexType>

<xs:element>
  <xs:complexType>
    <xs:attribute name="version" type="xs:string"
      use="required" fixed="1.1"/>
  </xs:complexType>
</xs:element>

<xs:group name="msmlRequest">
  <xs:choice>
    <xs:element name="createconference">
      <xs:complexType>
        <xs:all>
          <xs:element name="audioMix" type="audioMixType"/>
          <xs:element name="videoLayout" type="videoLayoutType"/>
          <xs:element name="reserve">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="resource" maxOccurs="unbounded">
                  <xs:complexType>
                    <xs:sequence>
                      <xs:any namespace="##other" processContents="lax"
                        minOccurs="0" maxOccurs="unbounded"/>
                    </xs:sequence>
                    <xs:attribute name="n" type="xs:positiveInteger" default="1"/>
                    <xs:anyAttribute namespace="##any"/>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:attribute name="required" type="boolean.datatype" default="true"/>
        </xs:all>
        <xs:attribute name="name" type="msmlInstanceID.datatype"/>
        <xs:attribute name="deletewhen" default="never"/>
      </xs:complexType>
    </xs:element>
  </xs:choice>
</xs:group>
<xs:simpleType>
  <xs:restriction base="xs:string"/>
  <xs:enumeration value="nomedia"/>
  <xs:enumeration value="nocontrol"/>
  <xs:enumeration value="never"/>
</xs:restriction>
</xs:simpleType>

<xs:attribute name="term" type="boolean.datatype" default="true"/>
<xs:attribute ref="mark"/>
</xs:complexType>
</xs:element>

<xs:element name="modifyconference">
  <xs:complexType>
    <xs:all>
      <xs:element name="audioMix" type="audioMixType" minOccurs="0"/>
      <xs:element name="videoLayout" type="videoLayoutType" minOccurs="0"/>
    </xs:all>
    <xs:attribute name="id" type="confID.datatype" use="required"/>
    <xs:attribute ref="mark"/>
  </xs:complexType>
</xs:element>

<xs:element name="destroyconference">
  <xs:complexType>
    <xs:all>
      <xs:element name="audioMix" type="basicAudioMixType" minOccurs="0"/>
      <xs:element name="videoLayout" type="basicVideoLayoutType" minOccurs="0"/>
    </xs:all>
    <xs:attribute name="id" type="confID.datatype" use="required"/>
    <xs:attribute ref="mark"/>
  </xs:complexType>
</xs:element>

<xs:element name="dialogstart">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="momlRequest"/>
      <xs:group ref="sendType" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="target" type="independentID.datatype" use="required"/>
    <xs:attribute name="type" type="dialogLanguage.datatype" use="required"/>
    <xs:attribute name="name" type="msmlInstanceID.datatype"/>?
    <xs:attribute name="src" type="xs:anyURI" use="optional"/>
    <xs:attribute ref="mark"/>
  </xs:complexType>
</xs:element>
<xs:element name="dialogend">
  <xs:complexType>
    <xs:attribute name="id" type="dialogID.datatype" use="required"/>
    <xs:attribute ref="mark"/>
  </xs:complexType>
</xs:element>

<xs:element name="join">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="stream" type="streamType" minOccurs="0" maxOccurs="4"/>
    </xs:sequence>
    <xs:attribute name="id1" type="independentID.datatype" use="required"/>
    <xs:attribute name="id2" type="independentID.datatype" use="required"/>
    <xs:attribute ref="mark"/>
  </xs:complexType>
</xs:element>

<xs:element name="modifystream">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="stream" type="streamType" maxOccurs="4"/>
    </xs:sequence>
    <xs:attribute name="mark" type="mark.datatype"/>
  </xs:complexType>
</xs:element>

<xs:element name="unjoin">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="stream" type="basicStreamType" minOccurs="0" maxOccurs="4"/>
    </xs:sequence>
    <xs:attribute name="id1" type="independentID.datatype" use="required"/>
    <xs:attribute name="id2" type="independentID.datatype" use="required"/>
    <xs:attribute ref="mark"/>
  </xs:complexType>
</xs:element>

<xs:element name="monitor">
  <xs:complexType>
    <xs:attribute name="id1" type="connID.datatype" use="required"/>
    <xs:attribute name="id2" type="independentID.datatype" use="required"/>
    <xs:attribute name="compressed" type="boolean.datatype" default="false"/>
    <xs:attribute ref="mark"/>
  </xs:complexType>
</xs:element>

<xs:element name="send">
  <xs:complexType>
    <xs:attribute name="event" type="msmlEvent.datatype" use="required"/>
  </xs:complexType>
</xs:element>
<xs:attribute name="target" type="msmlTarget.datatype" use="required"/>
<xs:attribute name="valuelist" type="xs:string"/>
<xs:attribute ref="mark"/>
</xs:complexType>
</xs:element>
</xs:choice>
</xs:group>
<xs:complexType name="basicStreamType">
<xs:attribute name="dir">
<xs:simpleType>
<xs:restriction base="xs:string">
<xs:enumeration value="to-id1"/>
<xs:enumeration value="from-id1"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name="media">
<xs:simpleType>
<xs:restriction base="xs:string">
<xs:enumeration value="audio"/>
<xs:enumeration value="video"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
</xs:complexType>
</xs:complexType>
<xs:complexType name="streamType">
<xs:complexContent>
<xs:extension base="basicStreamType">
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element name="gain">
<xs:complexType>
<xs:attribute name="amt" use="optional">
<xs:simpleType>
<xs:restriction base="xs:integer">
<xs:minInclusive value="-96"/>
<xs:maxInclusive value="96"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name="agc" type="boolean.datatype"/>
<xs:attribute name="tgtlvl" use="optional">
<xs:simpleType>
<xs:restriction base="xs:nonPositiveInteger">
<xs:minInclusive value="-40"/>
<xs:maxInclusive value="0"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
</xs:complexType>
</xs:element>
</xs:choice>
</xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:attribute name="maxgain" default="10">
  <xs:simpleType>
    <xs:restriction base="xs:nonNegativeInteger">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="40"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
</xs:complexType>
</xs:element>
<xs:element name="clamp">
  <xs:complexType>
    <xs:attribute name="dtmf" type="boolean.datatype"/>
    <xs:attribute name="tones" type="boolean.datatype"/>
  </xs:complexType>
</xs:element>
<xs:element name="visual"/>
</xs:choice>
<xs:attribute name="preferred" type="boolean.datatype"/>
<xs:attribute name="display" type="xs:string"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:complexType name="basicAudioMixType">
  <xs:attribute name="id" type="xs:string" use="required"/>
</xs:complexType>
</xs:complexType>
<xs:complexType name="audioMixType">
  <xs:complexContent>
    <xs:extension base="basicAudioMixType">
      <xs:all>
        <xs:element name="asn" minOccurs="0">
          <xs:complexType>
            <xs:attribute name="ri" type="posDuration.datatype"/>  
          </xs:complexType>
        </xs:element>
        <xs:element name="n-loudest" minOccurs="0">
          <xs:complexType>
            <xs:attribute name="n" type="xs:positiveInteger" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:all>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="basicVideoLayoutType">
  <xs:attribute name="id" type="xs:string" use="required"/>
  <xs:attribute name="type" type="xs:string" use="required" fixed="text/msml-basic-layout"/>
</xs:complexType>
</xs:complexType>
<xs:complexType name="videoLayoutType">
  <xs:complexContent>
    <xs:extension base="basicVideoLayoutType">
      <xs:choice>
        <xs:element name="selector">
          <xs:complexType>
            <xs:complexContent>
              <xs:extension base="selectorType">
                <xs:sequence>
                  <xs:element name="root" type="rootType"/>
                </xs:sequence>
              </xs:extension>
            </xs:complexContent>
          </xs:complexType>
        </xs:element>
        <xs:sequence>
          <xs:element name="root" type="rootType"/>
          <xs:choice maxOccurs="unbounded">
            <xs:element name="selector" minOccurs="0" maxOccurs="unbounded">
              <xs:complexType>
                <xs:complexContent>
                  <xs:extension base="selectorType">
                    <xs:sequence>
                      <xs:element name="region"/>
                    </xs:sequence>
                  </xs:extension>
                </xs:complexContent>
              </xs:complexType>
            </xs:element>
            <xs:element name="region" minOccurs="0" maxOccurs="unbounded">
              <xs:complexType>
                <xs:complexContent>
                  <xs:extension base="regionType">
                    <xs:sequence>
                      <xs:element name="region"/>
                    </xs:sequence>
                  </xs:extension>
                </xs:complexContent>
              </xs:complexType>
            </xs:element>
          </xs:choice>
        </xs:sequence>
      </xs:choice>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="regionType">
  <xs:attribute name="id" type="xs:string" use="required"/>
  <xs:attribute name="left" type="xs:positiveInteger"/>
  <xs:attribute name="top" type="xs:positiveInteger"/>
  <xs:attribute name="relativeSize">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:enumeration value="1/4"/>
    <xs:enumeration value="1/3"/>
    <xs:enumeration value="2/3"/>
    <xs:enumeration value="3/4"/>
    <xs:enumeration value="1"/>
  </xs:restriction>
</xs:simpleType>

<xs:attribute name="priority">
  <xs:simpleType>
    <xs:restriction base="xs:float">
      <xs:minInclusive value="0"/>
      <xs:maxExclusive value="1"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>

<xs:attribute name="title" type="xs:string"/>
<xs:attribute name="titleTextColor" type="xs:string"/>
<xs:attribute name="titleBackgroundColor" type="xs:string"/>
<xs:attribute name="borderColor" type="xs:string"/>
<xs:attribute name="borderWidth" type="xs:positiveInteger"/>
<xs:attribute name="logo" type="xs:anyURI"/>
</xs:complexType>

<xs:complexType name="selectorType">
  <xs:attribute name="id" type="xs:string" use="required"/>
  <xs:attribute name="method" use="required">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="vas"/>
        <xs:enumeration value="sequence"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="status" default="active">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="active"/>
        <xs:enumeration value="disabled"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="si" type="posDuration.datatype" default="1s"/>
  <xs:attribute name="blankothers" type="xs:boolean" default="false"/>
  <xs:attribute name="speakersees" default="current">
    <xs:simpleType>
      <xs:restriction base="xs:string">
      </xs:simpleType>
    </xs:attribute>
  </xs:attribute>
</xs:complexType>
<xs:enumeration value="current"/>
<xs:enumeration value="previous"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
</xs:complexType>
<xs:complexType name="rootType">
<xs:attribute name="size" default="CIF">
<xs:simpleType>
<xs:restriction base="xs:string">
<xs:enumeration value="16CIF"/>
<xs:enumeration value="4CIF"/>
<xs:enumeration value="CIF"/>
<xs:enumeration value="QCIF"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name="backgroundcolor" type="xs:string" default="black"/>
<xs:attribute name="backgroundimage" type="xs:anyURI"/>
</xs:complexType>
<xs:attribute name="mark" type="mark.datatype"/>
</xs:schema>

Following is the schema which defines the basic datatypes used by the other schema. Note that several regular expressions required them to be split across two lines for formatting reasons.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified"
attributeFormDefault="unqualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:simpleType name="conferenceType.datatype">
<xs:restriction base="xs:string">
<xs:enumeration value="audio.basic"/>
<xs:enumeration value="audio.advanced"/>
</xs:restriction>
</xs:simpleType>
<xs:simpleType name="msmlInstanceID.datatype">
<xs:restriction base="xs:string">
<xs:pattern value="[a-zA-Z0-9.:-_]*/"/>
</xs:restriction>
</xs:simpleType>
<xs:simpleType name="connID.datatype">
<xs:restriction base="xs:string">
<xs:pattern value="conn:[a-zA-Z0-9.:-]+"/>
</xs:restriction>
</xs:simpleType>
```
<xs:simpleType name="confID.datatype">
    <xs:restriction base="xs:string">
        <xs:pattern value="conf:[a-zA-Z0-9.:-_]+"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="operatorID.datatype">
    <xs:restriction base="xs:string">
        <xs:pattern value="conf:[a-zA-Z0-9.:-_]+"/>
        <xs:pattern value="conn:[a-zA-Z0-9.:-_]+"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="dialogID.datatype">
    <xs:restriction base="xs:string">
        <xs:pattern value="conf:[a-zA-Z0-9.:-_]+/dialog:[a-zA-Z0-9.:-_]+"/>
        <xs:pattern value="conn:[a-zA-Z0-9.:-_]+/dialog:[a-zA-Z0-9.:-_]+"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="independentID.datatype">
    <xs:restriction base="xs:string">
        <xs:pattern value="conf:[a-zA-Z0-9.:-_]+"/>
        <xs:pattern value="conn:[a-zA-Z0-9.:-_]+"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="duplex.datatype">
    <xs:restriction base="xs:string">
        <xs:enumeration value="half"/>
        <xs:enumeration value="full"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="confclass.datatype">
    <xs:restriction base="xs:string">
        <xs:enumeration value="standard"/>
        <xs:enumeration value="preferred"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="dialogLanguage.datatype">
    <xs:restriction base="xs:string">
        <xs:enumeration value="application/moml+xml"/>
        <xs:enumeration value="application/voicexml+xml"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="msmlEvent.datatype">
    <xs:restriction base="xs:string"/>
</xs:simpleType>

<xs:simpleType name="msmlEventName.datatype">
    <xs:restriction base="xs:string">
        <xs:pattern value="msml.dialog.exit"/>
        <xs:pattern value="msml.conf.asn"/>
    </xs:restriction>
</xs:simpleType>
<xs:pattern value="msml.conf.nomedia"/>  
<xs:pattern value="msml.dialog.exit"/>  
<xs:pattern value="[a-zA-Z0-9.-_]+"/>  
</xs:restriction>  
</xs:simpleType>  
<xs:simpleType name="msmlTarget.datatype">  
<xs:restriction base="xs:string">  
<xs:pattern value="conf:[a-zA-Z0-9.-_]+(/oper:[a-zA-Z0-9.-_]+|\*)*/">  
<xs:pattern value="conn:[a-zA-Z0-9.-_]+(/oper:[a-zA-Z0-9.-_]+|\*)+/">  
</xs:restriction>  
</xs:simpleType>  
<xs:simpleType name="msmlEventSource.datatype">  
<xs:restriction base="xs:string">  
<xs:pattern value="conf:[a-zA-Z0-9.-_]+"/>  
<xs:pattern value="("conf:[a-zA-Z0-9.-_]+|conn:[a-zA-Z0-9.-_]+)"/>  
</xs:restriction>  
</xs:simpleType>  
<xs:simpleType name="msmlEventNameValue.datatype">  
<xs:restriction base="xs:string"/>  
</xs:simpleType>  
<xs:simpleType name="mark.datatype">  
<xs:restriction base="xs:string">  
<xs:pattern value="[a-zA-Z0-9.-_]+"/>  
</xs:restriction>  
</xs:simpleType>  
</xs:schema>
14. Security Considerations

MSML is invoked through other languages and protocols and as such security considerations depend on those environments. MSML being an XML based language, security considerations as defined by RFC 3023 [9]. are applicable.
15. IANA Considerations

IANA registration for ’application/msml+xml’ media type is planned.

MGCP and Megaco/H.248 packages are registered with IANA so that there is no conflict between packages developed by different authors. The package scheme planned for future extensibility of MSML/MOML will also require IANA registry of packages.
16. Acknowledgements

Version -06 was derived from earlier versions co-authored by Tim Melanchuk (contact: "<tim.melanchuk@gmail.com>") and Garland Sharratt.

Adnan Saleem and Yong Xin of Convedia provided key insights, both theoretic and through development experience, on several earlier versions of the drafts. Stephen Buko and George Raskulinec of Intel made numerous valuable contributions for new feature additions to MSML and MOML versions -06. Gilles Compienne and Ben Smith, both of Ubiquity Software, provided feedback on several versions. Chris Boulton of Ubiquity, and Michael Rice of VocalData helped clarify several issues in the -00 draft, while Bruce Walsh and Kevin Fitzgerald, both of Spectel, provided important feedback on that draft. Peter Danielsen of Lucent has contributed thoughtful and detailed reviews for several versions of the draft.
17. References

17.1. Normative References


17.2. Informative References


Authors’ Addresses

Adnan Saleem
Convedia
4190 Still Creek Drive, Suite 300
Vancouver, BC V5C 6C6
Canada

Phone: +1 604 918 6376
Email: asaleem@convedia.com

Garland Sharratt
Convedia
4190 Still Creek Drive, Suite 300
Vancouver, BC V5C 6C6
Canada

Phone: +1 604 918 6393
Email: gsharratt@convedia.com
Intellectual Property Statement

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at http://www.ietf.org/ipr.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Disclaimer of Validity

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Copyright Statement

Copyright (C) The Internet Society (2005). This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.