Network Work Group
Internet Draft
Document: draft-vandyke-mscml-00.txt
Category: Informational
Expires: April 28, 2002

E. Burger (ed.)
J. Van Dyke
A. Spitzer
SnowShore Networks, Inc.

October 28, 2002

SnowShore Media Server Control Markup Language and Protocol

Status of this Memo

Abstract

Media Server Control Markup Language (MSCML) is a markup language used in conjunction with SIP to provide advanced conferencing and IVR functions.

Table of Contents

1. Conventions used in this document

In examples, "C:" and "S:" indicate lines sent by the client and server respectively.

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 [2].

2. Introduction

This document describes the SnowShore Media Server Control Markup Language (MSCML). This document describes payloads that one can send with a standard SIP INVITE to a media server. The document [3] describes media server SIP URI formats.

Prior to MSCML, there was not a standard way for the delivery of SIP-based enhanced conferencing. Basic SIP constructs, such as described in [3], serves simple n-way conferencing well. The SIP URI provides a natural mechanism for identifying a specific SIP conference, while INVITE and BYE methods elegantly implement conference join and leave semantics. However, enhanced conferencing applications also require features such as sizing and resizing, in-

conference IVR operations (e.g. recording/playing participant names to the full conference) and conference event reporting. MSCML payloads within standard SIP INVITE and INFO requests realize these features.

There are two broad classes of MSCML functionality. The first class includes primitives for advanced conferencing such as conference configuration, participant leg manipulation and conference event reporting. The second class comprises primitives for interactive voice response (IVR). These include playing audio, collecting digits, and recording audio.

The IVR features of MSCML originally evolved simply as an adjunct for conferencing. In many scenarios it was impractical or inconvenient to establish a dialog with a distinct IVR resource and then re-join the conference. However, MSCML works well for simple IVR such as prompt-and-collect for SIP Proxy Servers or Media Gateway Controllers. On the other hand, for complex IVR it may be more appropriate to employ a full IVR markup language such as VoiceXML [4].

In general, a media server offers services to SIP UAC's on application servers, feature servers, and media gateway controllers. See [5] for definitions of these terms. It is unlikely, but not prohibited, for end user SIP UAC's to have a direct signaling relationship with a media server.

This document describes a working framework and protocol with which there is considerable implementation experience. Application developers and service providers have created several MSCML-based services since the initial version was made available more than a year ago. This experience is highly relevant to the ongoing work of the IETF, particularly the SIP, SIPPING, and MMUSIC work groups.

3. Use of SIP Request Methods

As mentioned above, MSCML payloads may be carried in either SIP INVITE or INFO requests. The initial INVITE, which creates an enhanced conference, MUST include an MSCML payload. The initial INVITE, which joins a participant leg to an enhanced conference, MAY include an MSCML payload. All mid-call MSCML payloads are sent via SIP INFO requests.

MSCML responses are transported in the final response to the SIP INVITE containing the matching MSCML request or in a SIP INFO message. The only allowable final response to a SIP INFO containing a message body is a 200 OK (Per RFC 2976 [6]). Therefore, when the MSCML request is sent via SIP INFO the MSCML response is carried in a separate INFO request. In general, these responses are asynchronous in nature and require a separate transaction due to timing considerations.

There has been considerable debate on the use of the SIP INFO method for any purpose. Our experience is that MSCML would not have been possible without it. When MSCML was implemented the first SIP Event Notification draft had just been published. At that time, use of SUBSCRIBE/NOTIFY within an existing dialog was undefined. This prevented its use in MSCML since all events occurred in an INVITE established dialog. And while SUBSCRIBE/NOTIFY was well suited for reporting conference events its semantics seemed inappropriate for modifying a participant leg or conference setting where the only "event" was the success or failure of the request. Lastly, since SIP INFO was an established RFC it was well supported in all the SIP stack implementations available at that time. We had few if any interoperability issues as a result.

SIP has progressed incredibly quickly and we will need to reevaluate some of the decisions that resulted in the original design of MSCML. However, we can confidently say that the availability of a widely supported, flexible request method was very important to the development and adoption MSCML.

4. MSCML Usage and Design

To avoid undue complexity two rules were established regarding MSCML usage. The first is that only one MSCML body may be present in a SIP request. The second is that each MSCML body may contain only one request or response. This greatly simplified transaction management. MSCML syntax does provide for the unique identification of multiple requests in a single body part but this is not currently allowed.

5. Advanced Conferencing

The advanced conferencing model is a star controller model, with both signaling and media directed to a central location. Figure 1 depicts a typical signaling relationship between end users' UAC's, a conference application server, and a media server.


```
+-------+
| UAC 1 |---\   Public URI  +-------------+
+-------+    \ _____________| Application |
              /    /        |   Server    |     Not shown:
+-------+    /    /         +-------------+     RTP flows directly
| UAC 2 |---/    /                 | Private    between UAC's and
+-------+       /                  |   URI      Media Server
    .          /            +--------------+
    :         /             |              |
+-------+    /              | Media Server |
| UAC n |---/               |              |
+-------+                   +--------------+
```

Figure 1 - Conference Model

Each UAC sends an INVITE to a Public Conference URI. Presumably the Application Server publishes this URI, or it is an ad hoc URI. In any event, the Application Server generates a Private URI, following the rules specified by [3]. That is, the URI is of the form:

    sip:conf=UniqueID@ms.carrier.net

Where UniqueID is a unique conference identifier, and ms.carrier.net is the host name or IP address of the media server. There is nothing to prevent the UAC's from contacting the media server directly. However, one would expect the owner of the media server to restrict who can use media server resources.

As for basic conferencing, described by [3], the first INVITE to the media server with a UniqueID creates a conference. However, in advanced conferencing, the first INVITE includes a MSCML configure_conference payload. The MSCML payload conveys extended session parameters (e.g. number of participants) that are not readily expressed in SDP but must be known to allocate the appropriate resources.

The first dialog established for an enhanced conference has several useful properties and is referred to as the "conference control leg." The control leg is used for play or record audio operations to/from the entire conference and no RTP is expected on the conference control leg. Therefore, the application must send either no SDP or hold SDP (c=0.0.0.0) in the initial INVITE request. In addition, the lifetime of the conference is the same as that of its control leg. This ensures that the conference remains in existence even if one or more participant legs unintentionally leaves the conference.

The <configure_conference> tag has two attributes that control the resources the media server sets aside for the conference. The attributes are reservedtalkers and reserveconfmedia. Reservedtalkers sets the maximum number of talker legs. Reserveconfmedia, if set to "Yes", allocates resources for playing

or recording audio to or from the entire conference. The default for reserveconfmedia is "Yes".

The application server can include any MSCML command in the initial INVITE, with the exception of asynchronous commands, such as <play> or <record>. The application server must issue asynchronous commands separately (e.g., in INFO messages) to avoid ambiguous responses.

For example, to create a conference with up to 120 active talkers and the ability to play audio into the conference or record parts or all of the conference, the application server specifies both attributes, as shown in Figure 2.

```
<?xml version="1.0">
  <MediaServerControl version="1.0">
    <request>
      <configure_conference reservedtalkers="120"/>
    </request>
  </MediaServerControl>
```

Figure 2 - 120 Speaker MSCML Example

Figure 3 shows a conference with up to five active speakers without the capability to play or record audio into the conference.

```
<?xml version="1.0">
  <MediaServerControl version="1.0">
    <request>
      <configure_conference reservedtalkers="5"
                            reserveconfmedia="no"/>
    </request>
  </MediaServerControl>
```

Figure 3 - 5 Speaker MSCML Example

Once the application server has created the conference Control Leg, the server can join participants to the conference. Per [3], the application server directs the INVITE to the Private Conference URI described above. In the example given, this would be sip:UniqueID@ms.carrier.net .

Conference legs have a number of parameters the application server can modify. The defaults are as follows in Table 1. Following sections will discuss the meaning of the parameters in detail.

Table 1 - Conference Leg Parameters

Parameter	Default	Description
inputgain	auto	Use AGC to determine input gain for leg
outputgain	auto	Use AGC to determine output gain for leg
type	talker	Consider this leg's audio for mixing in the output mix
dtmfclamp	yes	Remove detected DTMF digit from audio
toneclamp	yes	Remove loud single-frequency tone from audio

If the default parameters are acceptable for the leg the application server wishes to enter into the conference, then a normal SIP INVITE is sufficient. However, if the application server wishes to modify one or more of the parameters, the application server can include a MSCML body in addition to the SDP body.

The application server can modify the conference leg parameters by issuing a SIP INFO on the selected dialog representing the conference leg. Of course, the application server cannot modify SDP in an INFO message.

To remove a leg from the conference, the application server issues a SIP BYE request on the selected dialog representing the conference leg.

The application server can terminate all legs in a conference by issuing a SIP BYE request on the Conference Control Leg. If one or more participants are still in the conference when the media server receives a SIP BYE request on the Conference Control Leg, the media server issues SIP BYE requests on all of the remaining conference legs to ensure clean up of the legs.

The media server returns a 200 OK to the SIP BYE request as it sends BYE requests to the other legs. This is because we cannot issue a provisional response to a non-INVITE request, yet the teardown of the other legs may "take a while".

Once the conference has begun, the application server can manipulate the conference as a whole by issuing commands on the Conference Leg. For example, the application server can request the media server to record the conference, play a prompt to the conference, change the input or output gain for the conference as a whole, and report on events. The elements for these commands are <playrecord>, <play>, <inputgain>, <outputgain>, and <subscribe>, respectively.

Figure 4 shows two sample commands. The first plays a prompt into the conference. The second records the entire conference to the URI specified by recurl over NFS.


```
<?xml version="1.0">
  <MediaServerControl version="1.0">
    <request>
      <play
promptурl="http://prompts.carrier.net/us_EN/welcome.au"/>
    </request>
  </MediaServerControl>


<?xml version="1.0">
  <MediaServerControl version="1.0">
```

```
        <request>
          <playrecord
    recurl="file://archive.carrier.net/conferences/archives/011208.au"
                        beep="no"
                        initsilence="-1" endsilence="-1" />
        </request>
      </MediaServerControl>
```

Figure 4 - Sample Full Conference Audio Commands

The response to this last request will be similar to Figure 5.

```
  <?xml version="1.0">
    <MediaServerControl version="1.0">
     <response request="playrecord" code="200" text="OK"/>
  </MediaServerControl>
```

Figure 5 - Sample Change Command Response

Later event reporting comes through SIP INFO messages. Figure 6 shows an example report.

```
     <?xml version="1.0">
       <MediaServerControl version="1.0">
         <notification>
           <conference uniqueID="ab34h76z" numtalkers="16"
                       numlisteners="1382">
             <activetalkers>
               <talker callID="myhost4sn123"/>
               <talker callID="myhost2sn456"/>
               <talker callID="myhost12sn78 />
             </activetalkers>
           </conference>
         </notification>
       </MediaServerControl>
```

Figure 6 - Active Talker Event Example

An application server can modify a leg by issuing an INFO on the dialog associated with the participant leg. For example, Figure 7 mutes a conference leg.

```
     <?xml version="1.0">
       <MediaServerControl version="1.0">
         <request>
```

```
          <configure_leg mixmode="mute"/>
        <request>
      </MediaServerControl>
```

Figure 7 - Sample Change Leg Command

In Figure 4 we saw a request to play a prompt to the entire conference. We can also request to play a prompt to an individual call leg. If we want to play a prompt or collect digits only on a single leg, we issue the commands within the dialog for the of the desired conference participant.

6. Interactive Voice Response (IVR)

In the IVR model, the Media Server acts as a media processing proxy for the UAC. This is particularly useful when the UAC is a media gateway or other device with limited media processing capability.

```
                              +--------------+
                 Service URI  | Application  |
              /---------------|    Server    |
             /(e.g., RFC3087) +--------------+
            /                        |  MSCML
           /                         | Session
          /                   +--------------+
  +-----+/       RTP          |              |
  | UAC |=====================| Media Server |
  +-----+                     |              |
                              +--------------+
```

Figure 8 - IVR Model

The IVR service supports basic Interactive Voice Response functions, playing announcements, collecting DTMF digits, and recording audio, based on Media Server Control Markup Language (MSCML) directives added to the message body of a SIP request.

Multifunction media servers SHOULD use the URI conventions described in [3]. For review, the IVR service indicator is "ivr":

sip:ivr@ms.carrier.net

One may carry the request payload for IVR in either the initial SIP INVITE or INFO requests.

Mid-call requests must use the INFO method. The INFO method reduces certain timing issues that occur with re-INVITES and also uses less processing on both the application server and Media Server.

The Media Server notifies the application that the command has completed through a <response> message containing final status information and data such as collected DTMF digits.

The media server does not queue IVR requests. If the media server receives a request while another is in progress, the media server stops the first operation and it carries out the new request. The Media Server generates a <response> message for the first request and returns any data collected up to that point. If an application wishes to stop a request in progress but does not wish to initiate another operation, it issues a <stop> request. This also causes the Media Server to generate a <response> message.

The Media Server treats a SIP re-INVITE with hold media (c=0.0.0.0) as an implicit <stop> request. The media server immediately terminates the running <play>, <playcollect> or <playrecord> request, and sends a <response>, indicating "reason=stopped".

6.1. Play Audio <play>

The application issues a <play> request to play an announcement without interruption and with no digit collection. One use, for example, is to announce the name of a new participant to the entire conference.

The application specifies the announcement to play by the prompt block in the body of the request.

Attributes include promptencoding (optional), which explicitly specifies the encoding (μ-law or a-law), and id (also optional). ID is an application-defined request identifier that correlates the asynchronous response with its original request and echoes back to the application in the Media Server's response.

When the announcement has finished playing, the Media Server sends a <response> payload to the application in a SIP INFO message.

The response may carry the id, the status code (e.g., 200), the status text (e.g., OK), and the reason (EOF or stopped).

6.2. Collect Digits <playcollect>

The application issues a <playcollect> request to optionally play an announcement and the collect digits.

This request has multiple attributes, all of which are optional.

The presence or absence of the prompt block controls whether there will be an announcement or the result of the request is to be digit collection only.

Whenever the media server receives a <playcollect> request, it will continuously buffer and examine collected digits. The media server compares previously buffered digits to the returnkey, escapekey, and maxdigits attributes to determine if any immediate action is required. This provides the type-ahead behavior for menu traversal and other types of IVR interactions.

The application may override type-ahead behavior by setting the cleardigits parameter to "yes", which removes all previously-buffered digits such that the only user input considered is what occurs after the request.

If cleardigits is set to "no", digits previously buffered will result in the prompt being barged immediately. Prompt play would never begin, and digit collection would start immediately.

The default for barge is "yes". If the barge attribute is set to "no", the cleardigits attribute implicitly has a value of "yes". This ensures that DTMF input occurring before the current collection is not left in the buffer after the request completes.

The application can set two special digits to invoke special processing when detected:

The escapekey, which defaults to *, indicates that the user intends to terminate the current operation without saving any input collected to that point. Detection terminates the request immediately and generates a response.

The returnkey, which defaults to #, indicates the user has completed input and wants to return all collected digits to the application. When the media server detects the returnkey, it immediately terminates collection and returns the collected digits to the application in the <response> message.

Several timer attributes control how long the Media Server waits for digits in the input sequence. All timer settings are in milliseconds.

o firstdigittimer controls how long the Media Server waits for the initial DTMF input before terminating collection.

o interdigittimer controls how long the Media Server waits between DTMF inputs.

o extradigittimer controls how long the Media Server waits for additional user input after the specified number of digits (linkblueparatextinkblue) have been collected.

The extradigittimer setting enables the "returnkey" input to be associated with the current collection. For example, if maxdigits is set to 3 and returnkey is set to #, the user may enter either "x#", "xx#" or "xxx#", where x represents a DTMF digit.

If the "returnkey" pattern is detected during the "extradigit" interval, the collected digits are returned to the application and the "returnkey" is removed from the digit buffer.

If this were not the case, the example would return "xxx" to the application and leave the terminating "#" in the digit buffer to be processed by the next <playcollect> request. This might result in the termination of the following prompt; clearly not what the user intended.

The extradigittimer has no effect unless returnkey has been set.

When the <playcollect> has finished playing, the Media Server sends a <response> payload to the application in a SIP INFO message.

The response may carry the id, the code (e.g., 200), the text(e.g., OK), the reason (match, timeout, returnkey, escapekey, or stopped), and the collected digits.

6.3. Recording Audio <playrecord>

The <playrecord> request directs the Media Server to capture the RTP it receives and deliver it to a URL specified by the controlling application.

This tag has multiple attributes. The required recurl attribute identifies the URL target for the recorded audio. All other attributes are optional.

The presence or absence of the prompt block controls whether or not a prompt plays before recording begins.

When the application requests the media server to prompt the caller before recording audio, <playrecord> has two stages. The first is equivalent to a <playcollect> operation. The application may set the prompt phase to be interruptible by DTMF input (barge) and may also specify an escape key that will terminate the <playrecord> request before the recording phase begins.

Detection of the escape key generates a response message, and the operation returns immediately. If any other keys are pressed and if the prompt has been set as interruptible (barge="yes"), then the play stops immediately and the recording phase begins.

Any digits collected in the prompt phase, with the exception of the recstopmask, are buffered and returned in the response.

If the request proceeds to the recording phase, any digits from the collect phase are discarded from the buffer to eliminate unintended termination of the recording.

The media server compares digits detected during the recording phase to the digits specified in the recstopmask to determine if they indicate a recording termination request.

The media server ignores digits not present in the recstopmask and passes them into the recording. If the recording is terminated because of a DTMF input, the collected digits are returned to the application in the <response>.

Once recording has begun, the media server writes the audio to the specified recurl URL no matter what DTMF events are detected. It is the responsibility of the application to examine the DTMF input returned in the <response> message to determine whether the audio file should be saved or if it should be deleted and potentially re-recorded.

Two attributes control how long the Media Server waits for the start of speech to begin the recording and the absence of speech to end the recording:

- o initsilence determines how long to wait for initial speech input before terminating (cancelling) the recording. This parameter may take an integer value in milliseconds, or may be set to -1, which directs the Media Server to wait indefinitely. The default is 3000 ms (3 seconds).

- o endsilence determines how long the Media Server waits after speech has ended to stop the recording. This parameter may take an integer value in milliseconds, or may be set to -1. With a value of -1, the recording will continue indefinitely after speech has ended and may terminate due to a DTMF keypress or because the maximum desired duration has been reached. The default value is 4000 ms (4 seconds).

If the endsilence timer expires, the Media Server trims the end of the recorded audio by an amount equal to the endsilence parameter.

Additional attributes are:

- o mode (whether the recording will overwrite or append).

- o reencoding (whether encoding is mu-law or a-law).

- o duration (time in ms for the entire recording.

- o beep (whether a beep will signify the start of recording).

When the recording is finished, the media server generates a <response> message and sends it to the application in a SIP INFO message. The response contains the id, the code (e.g., 200, 400, 501), the reason (e.g., digit, end_silence, init_silence, max_duration, escapekey, error, or stopped), collected digits, and the reclength (size of the recorded file in bytes).

6.4. Stop Request <stop>

The application issues a <stop> request when the objective is to stop a request in progress and not initiate another operation. This request generates a <response> message from the Media Server.

The only attribute is id, which is optional.

The application-defined request id correlates the asynchronous response with its original request and echoes back to the application in the Media Server's response.

The response may carry the id, the code (e.g., 200), and the text (e.g., OK).

Note that the Media Server treats a SIP re-INVITE with hold media (c=0.0.0.0) as an implicit <stop> request. The media server immediately terminates the running <play>, <playcollect> or <playrecord> request, and sends a <response>, indicating "reason=stopped".

6.5. Prompt Block <prompt>

This block in the body of the <play>, <playcollect>, or <playrecord> request contains one or more references to physical audio files, provisioned sequences, or variables that are played in the order in which they appear.

The following is a sample prompt block.

```
<prompt baseurl="file:////opt/snowshore/prompts/conf/">
        <audio url="please_enter.wav"/>
        <variable type="silence" value="1"/>
        <audio url="your.raw" encoding="a-law"/>
        <variable type="silence" value="1"/>
        <audio
           url="http://prompts.carrier.net/pin_number.wav"/>
</prompt>
```

The baseurl attribute is the base URL prepended to the URL attributes within the <prompt> block.

Each audio element in a <prompt> block refers to an audio file or provisioned sequence for the media server to play. The media server plays audio files in the order in which they are listed in the block.

7. Response Attributes and Return Codes

7.1. SIP

The Media Server acknowledges receipt of an application request by sending a response of either 200 OK or 415 BAD MEDIA TYPE. (The latter is sent when the SIP request contains a content type other than "application/sdp" or "application/mediaservercontrol+xml").

The <response> message is transported in a SIP INFO request.

If there is an error in the request or the request cannot be completed, the <response> message is sent very shortly after receiving the request. If the request is able to proceed, the <response> contains final status information as listed below.

7.2. HTTP

The Media Server processes the request and returns a <response> message in the body of the http POST.

7.3. <response> Attributes

If an ID was specified in the request, that id will be echoed back to the application in the response.

The "code" is the result code for the request. It can take the following values.

- o 200 indicates command completed.
- o 400 for <playrecord> indicates command not accepted due to an error. The text attribute describes the cause of the error.
- o 501 for <playrecord> indicates an error because the media server does not support the URL type specified.

The "digits" are the returned digits for <playcollect> and <playrecord>. Its value is the collected digits, if any.

The "reason" is why the command terminated. For all requests, the reason "stopped" indicates that a <stop> request, another command, or a re-INVITE with hold media stopped the request.

For the <play> request, the "EOF" reason means the media server played out to the end of the file.

For the <playcollect> request, "match" means a match was found; "timeout" means no digit was received before the time-out timer expired; "returnkey" and "escapekey" means the return key or escape key terminated the operation, respsectively; and "interrupted" means another request interrupted the <playcollect> request.

For the <playrecord> request, "digit" means a digit was detected; "end_silence" means the recording terminated because the trailing silence timer expired; "init_silence" means that no voice was detected; "max_duration" means the recording terminated because the maximum time for recording completed; "escapekey" means the user entered the escape key in either play or record mode, thus terminating the recording; or "error", for a general operation failure.

The "reclength" is the length of the recording in bytes for a <playrecord>.

The "text" is the descriptive text associated with the response code.

8. Formal Syntax

The following syntax specification uses the augmented Data Type Definition (DTD) as described in XML [7].

```
<?xml version="1.0"?>
<!-- ============================================================ -->
<!-- MediaServerControl Document Type Description                 -->
<!-- Copyright (c) 2001-2002 SnowShore Networks, Inc.             -->
<!-- All Rights Reserved                                          -->
<!-- SnowShore Networks Confidential and Proprietary Information -->
<!-- ============================================================ -->

<!ELEMENT MediaServerControl (request | response | notification)>
<!ATTLIST MediaServerControl version (1.0) #REQUIRED>

<!ELEMENT request (configure_conference | configure_leg | play |
                   playcollect | playrecord | stop)>

<!ELEMENT configure_conference (inputgain?, outputgain?,
                                subscribe?)>
<!ATTLIST configure_conference
     id CDATA #IMPLIED
     reservedtalkers CDATA #IMPLIED
     reserveconfmedia (yes | no) #IMPLIED>

<!-- Tags for gain control                                        -->
<!ELEMENT outputgain (auto | fixed)>
<!ELEMENT inputgain (auto | fixed)>
```



```
<!ELEMENT auto EMPTY>
<!ATTLIST auto
     startlevel CDATA #IMPLIED
     targetlevel CDATA #IMPLIED
     silencethreshold CDATA #IMPLIED>
```

```
<!ELEMENT fixed EMPTY>
<!ATTLIST fixed
     level CDATA #IMPLIED>

<!ELEMENT subscribe (events)>

<!ELEMENT events (activetalkers)>

<!ELEMENT activetalkers (talker+)?>
<!ATTLIST activetalkers
     report (yes | no) "no"
     interval CDATA #IMPLIED>
<!-- Acceptable values for interval range from 1-60 seconds      -->

<!ELEMENT talker EMPTY>
<!ATTLIST talker
     callid CDATA #REQUIRED>
<!-- The list of current talkers is used only when sending       -->
<!-- notifications to the calling application.  It should never  -->
<!-- be set when subscribing.                                     -->

<!ELEMENT configure_leg (inputgain?, outputgain?)>
<!ATTLIST configure_leg
     id CDATA #IMPLIED
     type (talker | listener) #IMPLIED
     mixmode (full | mute | preferred | parked) #IMPLIED
     dtmfclamp (yes | no) #IMPLIED>

<!-- Stops a play or record operation in progress                -->
<!ELEMENT stop EMPTY>

<!-- Plays an audio prompt, no barge-in or digit collection.     -->
<!-- <play/> generates a <response/> message when the specified  -->
<!-- prompt has finished playing or if an error occurs.          -->
<!ELEMENT play (prompt)?>
<!ATTLIST play
     id CDATA #IMPLIED
     prompturl CDATA #IMPLIED
     promptencoding (ulaw | alaw) #IMPLIED>
```

```
<!-- Plays an audio prompt, collects DTMF digits and returns the -->
<!-- digits to the application.  May also be used simply to      -->
<!-- collect digits if no sequence is specified.  <playcollect/> -->
<!-- sends an asynchronous <response/> message which is normally -->
<!-- generated when the desired digits have been collected or a  -->
<!-- timeout has expired.                                        -->
<!ELEMENT playcollect (prompt?, pattern?)>
<!ATTLIST playcollect
     id CDATA #IMPLIED
     prompturl CDATA #IMPLIED
     barge (yes | no) "yes"
     promptencoding (ulaw | alaw) #IMPLIED
     cleardigits CDATA "yes"
     maxdigits CDATA #IMPLIED
     firstdigittimer CDATA #IMPLIED
     interdigittimer CDATA #IMPLIED
     intdigcrittimer CDATA #IMPLIED
     extradigittimer CDATA #IMPLIED
     returnkey CDATA "#"
     escapekey CDATA "*">

<!-- <playrecord/> takes the audio from the associated session   -->
<!-- and records it to the location and format specified.  It    -->
<!-- generates a <response/> message if the request is in error, -->
<!-- when the recording session has been interrupted by DTMF,    -->
<!-- the specified duration has been exceeded or a timeout has   -->
<!-- expired.  The request has an optional prompt to be played   -->
<!-- prior to the start of recording.                            -->
<!ELEMENT playrecord (prompt)?>
<!ATTLIST playrecord
     id CDATA #IMPLIED
     prompturl CDATA #IMPLIED
     barge (yes | no) #IMPLIED
     cleardigits (yes | no) #IMPLIED
     escapekey CDATA "*"
     recurl CDATA #REQUIRED
     mode (append | overwrite) "overwrite"
     recencoding (ulaw | alaw) #IMPLIED
     initsilence CDATA #IMPLIED
     endsilence CDATA #IMPLIED
     duration CDATA #IMPLIED
     beep (yes | no) "yes"
     recstopmask CDATA "01234567890*#">

<!ELEMENT prompt (audio | variable)+>
<!ATTLIST prompt
     locale CDATA #IMPLIED
     baseurl CDATA #IMPLIED>
```

```
<!ELEMENT audio EMPTY>
<!ATTLIST audio
     url CDATA #REQUIRED
     encoding (ulaw | alaw) #IMPLIED>
<!-- The encoding attribute is required for files that are not in-->
<!-- self-describing .au or .wav format and do not have a well   -->
<!-- known extension (.ulaw).                                    -->

<!ELEMENT pattern (regex | digitmap)+>

<!ELEMENT regex EMPTY>
<!ATTLIST regex
    value CDATA #REQUIRED
    name CDATA #IMPLIED>

<!ELEMENT digitmap EMPTY>
<!ATTLIST digitmap
    value CDATA #REQUIRED
    name CDATA #IMPLIED>

<!ELEMENT variable EMPTY>
<!ATTLIST variable
    type (date | digit | duration | month | money | number |
          silence | string | time | weekday) #REQUIRED
    subtype (mdy | dmy | ymd | ndn | t12 | t24 | USD | gen | ndn |
             crd | ord) #IMPLIED
    value CDATA #REQUIRED>

<!ELEMENT response EMPTY>
<!ATTLIST response
     request (configure_conference | configure_leg | play |
              playcollect |playrecord | stop) #REQUIRED
     id CDATA #IMPLIED
     code CDATA #REQUIRED
     text CDATA #REQUIRED
     patternname CDATA #IMPLIED>

<!ELEMENT notification (conference)>

<!ELEMENT conference (activetalkers)>
<!ATTLIST conference
     uniqueid CDATA #REQUIRED
```

numtalkers CDATA #REQUIRED>

9. Security Considerations

Because media flows through a media server in a conference, the media server itself MUST protect the integrity, confidentiality, and security of the sessions. It should not be possible for a conference participant, on her own behalf, to be able to "tap in" to another conference without proper authorization.

Because conferencing is a high value application, the media server SHOULD implement appropriate security measures. This includes, but not limited to, access lists for application servers. That is, only a select list of application or proxy servers is allowed to create conferences, invite participants to sessions, etc. Note that the mechanisms for such security, like private networks, shared certificates, MAC white/black lists, are beyond the scope of this draft.

10. IANA Considerations

MSCML payloads are identified by the MIME type "application/mediaservercontrol+xml".

11. References

1 Bradner, S., "The Internet Standards Process -- Revision 3", BCP 9, RFC 2026, October 1996.
INFORMATIVE

2 Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
NORMATIVE

3 Van Dyke, J., et. al., "Basic Network Media Services with SIP", draft-burger-sipping-netann-02.txt, June 2002, work in progress.
INFORMATIVE

4 McGlashan, S. (ed.), "Voice Extensible Markup Language (VoiceXML) Version 2.0", W3C Working Draft, <http://www.w3.org/TR/voicexml20/>, 24 April 2002, work in progress.

INFORMATIVE

5 International Softswitch Consortium Reference Architecture, V1.2, http://www.softswitch.org, June 2002.
INFORMATIVE

6 S. Donovan, "The SIP INFO Method", RFC 2976, October 2000.
NORMATIVE

7 Bray, T. et. al., "Extensible Markup Language (XML) 1.0 (Second Edition)", W3C Recommendation, <http://www.w3.org/TR/REC-xml>, October 2000.
NORMATIVE

12. Contributors

The concept, development, documentation, and execution for MSCML was done by Jeff Van Dyke, Andy Spitzer, and Terence Lobo at SnowShore Networks, Inc. The IVR implementation was influenced by original work by Andy Spitzer while he was at The Telephone Connection, Inc.

Terence Lobo, Srinivas Motamarri, Haj Elfadil, and Edwina Nowicki contributed in being the first to eat what got cooked up.

13. Acknowledgments

The following individuals significantly assisted in the development, direction, or, most importantly, debugging of MSCML.

- o Gaurav Srivastva and Subhash Verma from BayPackets
- o Jon Hinckley from SkyWave/Sestro
- o Wesley Hicks, Ravindra Kabre, Kevin Summers from Sonus Networks
- o Diana Rawlins, Sharadha Vijay from WorldCom
- o Tim Wong from Z-Tel
- o Kevin Flemming for his feedback on the semantics of creation versus configuration for conferencing

The authors would like to thank Scotty Farber who made most of our techno-geek into English.

14. Author's Addresses

Jeff Van Dyke jvandyke@snowshore.com
Andy Spitzer aspitzer@snowshore.com
Eric Burger (Ed.) eburger@snowshore.com
SnowShore Networks, Inc.
285 Billerica Rd.
Chelmsford, MA 01824-4120
USA

Phone: +1 978/367-8400

Full Copyright Statement

Acknowledgement

The Internet Society currently provides funding for the RFC Editor function.

SnowShore Networks, Inc. is a member of the Internet Society.