INTERNET-DRAFT
Expires: December 1999
J. Ott/C. Perkins/D. Kutscher
Universitaet Bremen/UCL/Universitaet Bremen
June 1999

# The Message Bus: Messages and Procedures
draft-ott-mmusic-mbus-semantics-00.txt

## Status of this memo

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet- Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

## Abstract

In a variety of conferencing scenarios, a local communication channel is desirable for conference-related information exchange between co-located but otherwise independent application entities, for example those taking part in application sessions that belong to the same conference. In loosely coupled conferences such a mechanism allows for coordination of applications entities to e.g. implement synchronization between media streams or to configure entities without user interaction. It can also be used to implement tightly coupled conferences enabling a conference controller to enforce conference wide control within a end system.

The local conference Message Bus (Mbus) provides a means to achieve the necessary amount of coordination between co-located conferencing applications for virtually any type of conference. The Message Bus comprises two logically distinct parts: a message transport and addressing infrastructure and a set of common as well as media tool specific messages. This document defines protocol procedures for the Message Bus operation and the syntax and semantics of several sets of Mbus messages: common messages understood by all application entities as well as messages specific to particular classes of applications.

The underlying message passing and addressing mechanisms for the Mbus is defined in a companion Internet draft [3].

This document is a contribution to the Multiparty Multimedia Session Control (MMUSIC) working group of the Internet Engineering Task Force. Comments are solicited and should be addressed to the working group's mailing list at confctrl@isi.edu and/or the authors.

# 1. Introduction

## 1.1. Background

The requirement specification as defined in the companion requirements specification [2] provides a set of scenario descriptions for the usage of a local coordination infrastructure. The Message Bus defined in this and a companion document provides a suitable means for local communication that serves all of the purposes mentioned in the requirement draft.

## 1.2. Scope of this Document

Two components constitute the Message Bus: the (lower level) message passing mechanisms and the (higher level) messages and their semantics. While the basic transport mechanisms for the Mbus are defined in a companion Internet Draft [3], the purpose of this document is to define common as well as application-specific Mbus messages and their semantics. This includes

- definition of a simple naming scheme to allow for unambiguous command names as well as easy and conflict-free extensibility of the Mbus command set.
- definition of a set of mandatory Mbus management messages and associated procedures that enable the Mbus to function in virtually all kinds of conference settings;
- definition of a set of conditionally mandatory messages for conference control; and
- definition of several sets of optional messages for specific media types and/or functions.

The main body of this document addresses the first three bullet items thereby providing the foundation for the operation of the Mbus. The Mbus messages specific to particular protocols, media, and functions are contained in independent appendices to this document.

## 2. Command Naming Scheme

The general command syntax is described in the companion Mbus protocol Internet draft[3]. Command names SHALL be constructed using hierarchical names to group conceptually related commands under a common hierarchy. The delimiter between names in the hierarchy is ``.'' (dot).

The Mbus addressing scheme defined in [3] provides for specifying incomplete addresses by omitting certain elements of an address element list, enabling entities to send commands to a group of Mbus entities. Therefore all command names must be unambiguous in a way that it is possible to interpret or neglect them without considering the message's address.

A set of commands within a certain hierarchy that MUST be understood by every entity is defined in section 4 ("Entity Control"). Table 1 lists the pre-defined command prefixes:

| Command prefix | Description of command class |
|---|---|
| mbus. | General Basic Mbus commands |
| conf. | Commands related to conference control |
| rtp. | RTP-related commands |
| audio. | Commands specific to audio tools/engines |
| video. | Commands specific to video tools/engines |
| security. | Security-related commands |
| status. | Commands to communicate status information, error conditions etc. |

Table 1: Naming conventions for Mbus commands

In addition, tool specific commands have to be defined as well thus allowing each Mbus entity to define and use a number of private commands. All such commands must begin with the sequence

tool.<tool-name>,

for example tool.rat.

The following sections define the mentioned command classes.

## 3. Basic Mbus Operation and Management

### 3.1. Requirements

Before components of a conferencing system can communicate with one another using the Mbus, they need to mutually find out about their

existence. After this bootstrap procedure that each Mbus entity goes through all other entities listening to the same Mbus know about the newcomer and the newcomer has learned about all the other entities.

In order to minimize the dependencies of applications on one another and on the environment they are operating in, a bootstrap procedure must take into account that

- Mbus entities may be started in an arbitrary order
  - manually by the user from a command line interpreter (possibly with provision a set of command line parameters),
  - manually by the user from a window manager menu (i.e. in contrast to the former without explicit parameterization),
  - automatically from a conference-aware tool (such as SDR), or
  - initially at system startup (and thus may be listening in the background).[1]
- some Mbus entities may depend on continued existence of other Mbus entities or need to synchronize with them before being able to perform their functions properly; and
- a local coordination entity -- if present -- may take over control of the tools, but cooperation via the Mbus works as well if no such controller is present.

## 3.2. Basic Mechanisms

From the aforementioned requirements, the following mechanisms are devised as being necessary:

1. Self-announcement messages

   Any Mbus entity is supposed to announce its presence (on the Mbus) after starting up. This is to be done repeatedly throughout its lifetime to address the issues of startup sequence.

2. Alive messages

---

[1] Note that the main distinction between these various ways of starting application entities are a) the amount of configuration information passed to the application as command line options and b) whether applications are single-session or multi-session capable (i.e. whether or not a single application process is able to act as multiple instances on the Mbus or not).

Any Mbus entity should frequently indicate that it is still alive. This mechanism may be combined with the aforementioned self-announcement.

3. Synchronization messages

Any Mbus entity should be able to indicate that it is waiting for a certain event to happen (similar to a P() operation on a semaphore but without creating external state somewhere). In conjunction with this, an Mbus entity should be capable of indicating to another entity that this condition is now satisfied (similar to a semaphore's V() operation).

An appropriate set of commands that implements this conceptual specification is presented in the following section.

## 3.3. Mbus Management Protocol

### 3.3.1. Mbus Message Syntax and Procedures

The following Mbus messages are defined to implement the mechanisms described above:

HELLO

Syntax: mbus.hello ()

Parameters: - none -

Each Mbus entity MUST send HELLO messages after startup to the global Mbus channel. After transmission of the HELLO message, it shall start a timer after the expiration of which the next HELLO message shall be transmitted. The timer shall be set to a random value t_hello <= t <= t_hello + t_dither to avoid synchronization of HELLO messages. Transmission of HELLO messages MUST NOT be stopped unless the entity detaches from the Mbus.

HELLO messages are sent unreliably to all Mbus entities.

Each Mbus entity learns about other Mbus entities by observing their HELLO messages and tracking the sender address of each message.

The HELLO message is also used to track the liveness of any Mbus entity. Whenever an Mbus entity has not heard for a time span of n_dead*(t_hello+t_dither) from another Mbus entity it may consider this entity to have failed (or have quit silently). Note that no need for any action is necessarily implied from this observation.

BYE

Syntax: mbus.bye ()

Parameters: - none -

An Mbus entity that is about to terminate (or ``detach'' from the Mbus) announces this by transmitting a BYE message.

The BYE message is sent unreliably to all receivers.

WAITING

Syntax: mbus.waiting (condition)

Parameters: symbol condition
The condition parameter is used to indicate that the entity transmitting this message is waiting for a particular event to occur.

The WAITING messages may be broadcast to all Mbus entities, multicast an arbitrary subgroup, or unicast to a particular peer. Transmission of the WAITING message MUST be unreliable and hence has to be repeated at an application-defined interval (until the condition is satisfied).

If an application wants to indicate that it is waiting for several conditions to be met, several WAITING messages are sent (possibly included in the same Mbus payload). Note that HELLO and WAITING messages may also be transmitted in a single Mbus payload.

Appendix D presents a tool configuration scheme that allow tools to be parameterized on the command line in order to start them in ``waiting mode''.

GO

Syntax: mbus.go (condition)

Parameters: symbol condition
This parameter specifies which condition is met.

The GO message is sent by an Mbus entity to ``unblock'' another Mbus entity -- the latter of which has indicated that it is waiting for a certain condition to be met. Only a single condition can be specified per GO message. If several conditions are satisfied simultaneously multiple GO messages MAY be combined in a single Mbus payload.

The GO message MUST be sent reliably via unicast to the Mbus entity

to unblock.

QUERY

Syntax: mbus.query (variable)

Parameters: symbol variable
This parameter specifies the variable name.

The QUERY message is a general mechanism for obtaining arbitrary information from Mbus entities. The semantics and the variable names are application specific. QUERY messages are answered with INFO messages (see below).

The QUERY message can be multicast or sent reliably via unicast to a single Mbus entity or a group of entities.

INFO

Syntax: mbus.info (variable value)

Parameters: symbol variable
This parameter specifies the variable name.

string value
The variable parameter specifies the variable name requested in a QUERY messages and the value parameter, that can be of an arbitrary type, provides the requested information.

The INFO message is a general mechanism for delivering arbitrary information to Mbus entities. The semantics and the variable names are application specific.

The INFO message can be multicast or sent reliably via unicast to a single Mbus entity or a group of entities.

POLL

Syntax: mbus.poll (question alt_c choice_c (alternatives) context)

Parameters: symbol question
This parameter question specifies the question topic a vote is requested for.

Integer alt_c

Integer choice_c

List (alternatives)
A sequence of parameters specifying how to process and answer the question in the following format: (n k (c1 ... cn))

| | |
|---|---|
| n | Number of alternatives |
| k | Number of allowed choices |
| (c1 ... cn) | A List of alternatives (type string) |

Table 2: Specification parameters of a POLL message

list context
The context parameter contains application specific context information required for deciding on the question.

The POLL message is a general mechanism for requesting decisions on arbitrary questions from Mbus entities. The semantics, the question names and the alternatives are application specific.

The POLL message can be multicast or sent reliably via unicast to a single Mbus entity or a group of entities.

It is up to the inquiring application to ensure that

a) all desired entities are inquired (and reached) and

b) all responses are collected

VOTE

Syntax: mbus.vote (question (choice-list))

Parameters: symbol question
This parameter question specifies the question topic this vote refers to.

List choice-list
A list of selected choices that must conform to the specification that was received in a POLL message before. The list elements are of type Integer and represent indices to the alternatives-list of the POLL message starting from 0.

The VOTE message is a general mechanism for answering POLL messages. A entity can have a group of other entities decide on a question, collect the results as VOTE messages and evaluate these to infer further actions. See appendix H for a sample POLL/VOTE conversation scenario.

The VOTE message can be multicast or sent reliably via unicast to a

single Mbus entity or a group of entities.

### 3.3.2. Timers and Counters

The following values for timers and counters used in this document shall apply.

| Timer / Counter | Value |
|---|---|
| t_hello | 1 second |
| t_dither | 100 milliseconds |
| n_dead | 5 |

Table 2: Timer and counter values

As the Mbus is designed for a local system architecture it is not considered necessary to provide dynamic adaptation of these timers and counters to the number of Mbus entities.

## 4. Conference Control

The conference control part of the Mbus messages is intended to

a) provide a means for obtaining information about capabilities from other local application entities -- for both using this information for capability negotiation with other end systems and determining which commands are understood by another application entity --;

b) allow dynamic (re-)configuration of application entities with respect to various session parameters;

c) forward conference state changes (potentially negotiated by a local conference controller through a horizontal control protocol) to all other application entities; and

d) provide means for controlling call control entities, such as SIP or H.323 engines; and

e) allow application entities (in tightly controlled conferences) to request invocation of conference control services from a conference controller (within whatever system and conference policy constraints apply).

Commands of the ''Conference Control'' class SHALL be prefixed with ''conf.''. Table 3 lists the currently defined prefixes under the conf hierarchy.

| Command prefix | Description of command class |
|---|---|
| conf.cap. | Capability commands |
| conf.transport. | Media transport configuration commands |
| conf.call-control. | Call control commands |

Table 3: Naming conventions for Mbus conference control commands

## 4.1. Capabilities

In order to enable tightly controlled conferences a conference controller -- potentially the local coordination entity -- needs to determine not only which application entities are present but also which capabilities they have, in which application sessions they participate, and so forth. This leads to the following:

Each Mbus entity should support a capability query and respond by providing the requested information to the querier. The query may be asking for all capabilities of the queried entity or for a particular subset specified in the query. Upon receipt of such a query the inquired Mbus entity provides successively all the desired capability information, possibly after recursive queries from the querier.

Capability commands are to be defined at a later time.

## 4.2. Media transport configuration

The hierarchy conf.transport. contains commands for configuring media transport parameters of entities such as IP addresses, port numbers, ttl. These commands SHOULD generally not be sent to entity groups because each entity will require a unique parameter set.

### 4.2.1. address

Syntax: conf.transport.address (ipaddr port [cport])

Parameters: string ipaddr

list of integer port-list

integer ttl

The IP unicast or multicast address and associated port number(s) to be used for information transmission (and, in case of multicast) reception. For application entities using RTP, the port for RTCP addresses may be specified as the second element of the port list. TTL values can be specified with the ttl parameter.

## 4.3. Miscellaneous Conference Control

A conference controller that cannot map or translate every conference control related state transition in a tightly coupled conference to a entity specific remote control Mbus message will want to ''forward'' this information to the group of all Mbus entities in order to allow each entity to interpret this information independently. The objective is to define a set of generic Mbus commands that allows to represent the change of conference global variables without necessarily using these commands to remote-control entities explicitly.

Conference control related commands are prefixed with ''conf.''. Table 4 lists the prefixes under the conf hierarchy.

| Command prefix | Description of command class |
|---|---|
| conf.name | Set conference/session name |
| conf.call-control | Commands for signaling invitations etc. |
| conf.floor | Floor control commands/indications |
| | |
| conf.member | Membership lists |
| conf.chair | Conference chair indications |

Table 5: Naming conventions for Mbus control.conf commands

### 4.3.1. Conference parameters

A set of commands is defined to communicate conference parameters like conference/session names etc.

#### 4.3.1.1. Name

Syntax: conf.name (conference-name [session-name])

Parameters: conference-name
session-name
Sets the conference (and session) name for a conference.

## 4.4. Call Control Messages

Call control messages are intended for interaction with call control and invitation protocols such as H.323 and SIP. They are designed to constitute the union of the call control messaging needed by endpoints, gateways, proxies, multipoint controllers, and gatekeepers. This allows the use of the Message Bus as a gluing mechanisms to create any type of system from roughly the same building blocks.

Mbus call control messages are based on a common basic message set defined in the following that shall be supported by any kind of call control protocol entity. The basic message set may be augmented by protocol-specific extensions required for protocol specific interactions between a local controller and/or local applications on one side and the respective protocol engine on the other.

The following prefixing conventions apply for call control messages:

| Command prefix | Description of command class |
|---|---|
| conf.call-control. | basic call control message set |
| conf.call-control.h323. | extensions for H.323-specific call control messages |
| conf.call-control.sip. | extensions for SIP-specific call control messages |

Table 6: Call control message prefixes

### 4.4.1. The Basic Call Control Message Set

The basic set of messages is defined to provide the core functionality of initiating a call on one side, accepting or refusing it on the other, and providing progress information as well as allowing termination of the call on either side. The basic call control message set MUST be supported by any call control engine.

These messages are exchanged using unicast addressing between some local controlling entity and a call control engine implementing a call control or initiation protocol such as H.323 or SIP:

- Outgoing calls may be initiated by any local entity; the call control engine has to keep track of the initiator of a particular call and return all responses or events relating to this call to this entity -- which may be different on a per call basis. If the call control engine notices that the controlling entity for a particular call has gone (e.g. because the Mbus reliability mechanism indicates non-delivery of a call control message or a BYE message was seen from this entity), these messages are forwarded to the local controller. If no local controller is available, the call SHOULD be terminated.

- Indications about incoming calls are always forwarded to the local controller. If no local controller is present incoming calls SHOULD automatically be rejected by the call control engine.

All messages of the basic call control message set are sent reliably via unicast to the call control engine.

In this first draft, the definition of the basic call control message set is deliberately kept very much restricted:

- In this revision of the document, only the messages for establishing a simple (point-to-point) call are specified, along with very few messages dealing with supplementary services. Further Mbus messages supporting supplementary services are TBD.
- No explicit consideration is given so far for devices other than end systems -- although a simple gateway could probably be built based upon the present subset.

The following messages are specified so far:

CALL

The CALL message is sent to the call control engine to make the engine initiate a call to another endpoint using the parameters specified as part of the CALL message.

Syntax: conf.call-control.call (ref upi address-list call-type gw-proxy-list media-list status)

Parameters: string ref
A unique identifier for the call. This reference MUST be used for all further interactions relating to this between the call control engine and the initiating entity. Every newly created call identifier MUST be composed of the Mbus address of the creating entity and a second entity specific part in order to ensure uniqueness.

string upi
A universal personal identifier for the callee.

list of string address-list
An ordered list of transport addresses, alias names, UCIs, or phone numbers -- as indicated by the prefix preceding each address to be called. It is assumed that all these addresses refer to the same user, and only a single call will be established. The order in which the addresses are specified indicates a preference and contacting the target SHOULD be tried in that order.

symbol call-type
Indicates the intention of the call: join a conference

(or an n-way call), invite another user into a conference or an n-way call, or create a new call or conference.

list of string gw-proxy-list
An ordered list of (ordered) lists identifying proxies or gateways to be used for call setup if they are known. The n-th element in the list is a list of alternative gateways/proxies to be used in the n-th step in the call setup process. The gw-proxy-list may be empty.

list of string media-list
A list of media along with the preferred capability descriptions to be used for this particular call. If the same media type (e.g. audio) occurs repeatedly in the list (e.g. with different codecs) the relative order of the media descriptions indicates a preference (e.g. of one codec over the other).

Currently, media-list are SDP specifiations, but a new general format will be specified in the next revision of this draft.

symbol status
A parameter used to indicate certain attributes of a call process. This list is a subset of the following list of symbols:

| Symbol | Description |
|---|---|
| complete | Means that this call command contains all required information and that it can be processed immediately by a receiving entity. |
| partial | This call command is part of a series of call commands related to a single call. |
| final | This is the last call command in a series of call commands related to a single call. |

Table 7: Status symbols in a conf.call-control.call command

The symbols complete, partial, and final SHALL be used exclusively.

DISCONNECT

The DISCONNECT command is sent by the local controller to the call control engine to indicate that the specified call is to be disconnected. It can also be used by the local controller to inform the call control engine that a call has already been terminated by out-of-band communication, e.g. a horizontal conference control protocol. In this case a special (yet to be defined) reason code has to be passed with the command.

Syntax: conf.call-control.disconnect (ref reason)

Parameters: string ref
Identifies the call the DISCONNECT command refers to.

string reason
Indicates why the call was terminated. The reason will be set to user-initiated if the user simply ''hung up`` the phone. Other reason codes will be imported from H.323 and SIP. They are to be defined.

RINGING

The RINGING message is sent by the call control engine to the entity it received the corresponding CALL message from. RINGING indicates that one or more addresses at the far end were contacted and are now alerting the user.

Syntax: conf.call-control.ringing (ref (address-list))

Parameters: string ref
Identifies the call the RINGING message refers to.

list of string address-list
An ordered list of transport addresses, alias names, UCIs, or phone numbers -- as indicated by the prefix preceding each address to be called. It is assumed that all these addresses refer to the same user, and only a single call will be established. The order in which the addresses are specified indicates a preference and contacting the target SHOULD be tried in that order.

CONNECTED

The CONNECTED message is sent by the call control engine to the entity that initiated the call (on the calling side) and to the local controller (on the called side) to indicate that the call was successfully established.

Syntax: conf.call-control.connected (ref peer-address (media-list))

Parameters: string ref
Identifies the call the CONNECTED message refers to.

string peer-address
Indicates the (transport/alias/UCI/etc.) address of the peer endpoint (or a proxy/gateway/gatekeeper hiding its actual identity) that the call was finally established with.

list of string media-list
A list of media along with the capability descriptions that were initially negotiated for this particular call.

REJECTED

The REJECTED message is sent by the call control engine to the entity that initiated the call (on the calling side) and to the local controller (on the called side) to indicate that the call was rejected.

Syntax: conf.call-control.rejected (ref ((address reason)-list))

Parameters: string ref
Identifies the call the REJECTED message refers to.

list of list of string ((address reason)-list)
The (address reason) pair specifies which target address has rejected the call for which reason. As several addresses may have been tried explicitly, a list of addresses is returned, each paired with its particular rejection reason and possibly associated parameters. The details of the reason codes are to be defined; they are to be derived from H.323 and SIP and will include e.g. busy no-answer user-reject no-resources and authentication-failure among many others.

DISCONNECTED

The DISCONNECTED message is sent by the call control engine to the entity that initiated the call (on the calling side) and to the local controller (on the called side) to indicate that the call was disconnected.

Syntax: conf.call-control.disconnected (ref reason)

Parameters: string ref

Identifies the call the DISCONNECTED message refers to.

symbol reason
Indicates why the call was terminated. The reason will be set to user-initiated if the user simply ''hung up`` the phone. Other reason codes will be imported from H.323 and SIP. They are to be defined.

INCOMING CALL

The INCOMING CALL messages is sent by the call control engine to the local controller to indicate a call request from another endpoint.

Syntax:
conf.call-control.incoming-call (ref src-address (address-list) call-type (gw-proxy-list) (media-list))

Parameters: string ref
A unique identifier for the call. (See the notes concerning unique addresses at the description of the CALL command.) This reference MUST be used for all further interactions relating to this between the call control engine and the local controller entity.

string src-address
The address (transport address, alias name, UCI, phone number, etc.) of the endpoint initiating the call.

list of string address-list
An ordered list of transport addresses, alias names, UCIs, or phone numbers as sent by the calling endpoint with semantics similar to the address-list in the CALL message.

symbol call-type
Indicates the intention of the call; again, similar to the CALL message.

list of string gw-proxy-list
An ordered list of (ordered) lists identifying proxies or gateways to be used for call setup if they are known. Similar to the CALL message.

list of string media-list
A list of media along with the preferred capability descriptions proposed by the calling endpoint to be used for this particular call. If the same media type (e.g. audio) occurs repeatedly in the list (e.g. with different codecs) the relative order of the media descriptions indicates a preference (e.g. of one codec over the other).

ACCEPT

An ACCEPT message is sent by the local controller to the call control engine that has indicated an INCOMING CALL to indicate acceptance of the call.

Syntax: conf.call-control.accept (ref (media-list))

Parameters: string ref
Identifies the call the ACCEPT message refers to.

list of string media-list
A list of media along with the preferred capability descriptions selected by the local controller. This SHOULD be a strict subset of the media descriptions the calling endpoint has proposed for this particular call.

REJECT

A REJECT message is sent by the local controller to the call control engine that has indicated an INCOMING CALL to indicate rejection of the call.

Syntax: conf.call-control.reject (ref reason [params])

Parameters: string ref
Identifies the call the RINGING message refers to.

symbol reason
The reason code indicates why the call attempt was refused by the callee.
The details of the reason codes are to be defined; they are to be derived from H.323 and SIP and will include e.g. busy no-answer user-reject no-resources and authentication-failure among many others.

params Additional parameters may be provided along with the reason code. TBD.

REDIRECT

The REDIRECT command is sent by the local controller to the call control engine to indicate that the specified call is to be redirected to another specified address.

Syntax: conf.call-control.redirect (ref upi address-list attr)

Parameters: string ref
Identifies the call the REDIRECT command refers to.

string upi
A universal personal identifier for the callee.

list of string address-list
List of addresses where the call should be redirected to.

symbol attr
A symbol with the value ''temporarily'' or ''permanently'', signaling whether the redirection is temporarily or not.

REDIRECTED

The REDIRECTED command is sent by a call control engine to the local controller to indicate that the specified call has been redirected to the specified address. The default semantics in this case are that the call control engine can purge any state related to that call and the local controller has to decide on further actions. In case the redirection should be obeyed the local controller can initiate a new call by sending the CALL command with the address parameter obtained from the REDIRECTED command.

Call control engines that can decide themselves what to do after the reception of a protocol specific redirection can signal this by setting the status parameter to ''ACTIVE''. The semantics in this case are that the call control engine performs any required protocol specific action autonomously and that it will send the usual call setup related commands (CONNECTED, REJECTED etc.) during the course of the call setup. The local controller can terminate the call at any time with a DISCONNECT command. This behaviour would have to be configured by out of band means; the default behaviour is that the local controller is responsible for any reaction on REDIRECTED commands (signalled by setting status to PASSIVE).

Syntax: conf.call-control.redirected (ref upi addr-list attr status)

Parameters: string ref
Identifies the call the REDIRECT command refers to.

string upi
A universal personal identifier for the callee.

list of string addr-list
Address where the call should be redirected to.

symbol attr
A symbol with the value ''temporarily'' or

''permanently'', signaling whether the redirection is temporarily or not.

symbol status
One of ACTIVE and PASSIVE. Used to signal whether a call control engine performs the redirection iself or not.

FORWARD

The FORWARD command is sent by the local controller to the call control engine to indicate that the specified incoming call is to be forwarded to another (optionally specified) address. The second parameter is a list of strings that are to be interpreted as addresses. This list can be empty. The presence of elements in the address list denotes that the call control engine should use the specified address instead of determining it independently. If no addresses are provided the call control engine is requested to determine an address independently.

The FORWARD command can be used instead of REDIRECT when the end system acts as a application layer proxy that decides which calls are allowed to be forwarded. The forwarding can either happen with the call control protocol's implicit semantics (e.g. SIP forwarding) or the controller can explicitely specify the forwarding address.

The call control engine will send CONNECTED or REJECTED responses to inform the local controller of the result of the forwarding process.

Syntax: conf.call-control.forward (ref addr-list)

Parameters: string ref
Identifies the call the FORWARD command refers to.

list of string addr-list
List of (optional) address specifying where the call should be forwarded to.

FORWARDED

The FORWARDED command is sent by the call control engine to the local controller to indicate that the specified call has been forwarded to the specified address. The local controller can decide whether the call setup should continue or be interrupted (by sending a DISCONNECT command).

Syntax: conf.call-control.forwarded (ref addr-list)

Parameters: string ref
Identifies the call the FORWARD command refers to.

list of string addr-list
List of (optional) address specifying where the call has been forwarded to.

RELAYED

The RELAYED command is sent by the local controller to the call control engine to indicate that the specified incoming call is being forwarded to the specified address via another call control engine (e.g. from another protocol of administrative domain).

One scenario would be a incoming call that has been signalled by a SIP call control engine with a INCOMING-CALL command and that is then relayed to the H.323 call control engine. The local controller would inform the SIP engine by sending the RELAYED command.

A notation for address specifications (h323:bla) needs to be defined.

Syntax: conf.call-control.relayed (ref addr)

Parameters: string ref
Identifies the call this command refers to.

string addr-list
The address specifying where the call has been relayed to.

Further messages deemed necessary for the basic call control message set include the following

o PROGRESS

The details of these as well as further messages are to be defined.

4.4.2. Running a conference

Floor control, chairperson, policy, membership lists, etc.

4.4.2.1. Floor control

owner

Syntax: conf.floor.owner (cname)

Parameters: cname

The cname parameter designates the participant who the floor is currently assigned to.

need

Syntax: conf.floor.need ()

Parameters: --
Used to indicate that an application entity is requesting the floor for its media session.

release

Syntax: conf.floor.release ()

Parameters: --
Used to indicate that an application entity is releasing the floor it formerly requested for its media session.

4.4.2.2. Membership Lists

Commands starting with conf.member are intented for general messages on membership information that are not bound to specific transport protocols such as RTP[4].

Syntax: conf.member.list ([cname]*)

Parameters: cname
The cname parameter designates the current list of members.

Syntax: conf.member.add (cname)

Parameters: cname
The cname parameter designates a member that has joined the session.

Syntax: conf.member.remove (cname)

Parameters: cname
The cname parameter designates a member that has left the session.

4.4.2.3. Conference Chairs

Syntax: conf.chair (cname)

Parameters: cname
The cname parameter designates the current conference chair.

## 4.5. Status commands

The following table lists a few commands for generic status and error reports:

| Command | Description of command class |
|---|---|
| status.error | Error messages |
| status.warn | Warnings |
| status.info | Informational messages |

Table 8: Mbus Status commands

All status commands MUST be used with at least one string parameter that contains information on the status reported. Status commands MAY be used with arbitrary command prefixes. This allows classification of status commands where appropriate. E.G. status commands that are related to call-control MAY be prefixed with conf.call-control allowing an receiving entity to treat the command as a call-control specific status command. By convention status commands may have additional application specific parameters that are only useful if the concrete application context is known from the command prefix.

All status messages SHOULD be sent unreliably to all entities.

# 5. Authors' Addresses

Joerg Ott <jo@tzi.org>
Universitaet Bremen, TZI, MZH 5180
Bibliothekstr. 1
D-28359 Bremen
Germany
voice +49 421 201-7028
fax +49 421 218-7000

Colin Perkins <c.perkins@cs.ucl.ac.uk>
Department of Computer Science
University College London
Gower Street
London WC1E 6BT
United Kingdom

Dirk Kutscher <dku@tzi.org>
Universitaet Bremen, TZI, MZH 5160
Bibliothekstr. 1
D-28359 Bremen
Germany
voice +49 421 218-7595
fax +49 421 218-7000

## 6. References

[1] S. Bradner, ''Key words for use in RFCs to Indicate Requirement Levels'' RFC 2119, March 1997

[2] J. Ott, C. Perkins, and D. Kutscher, ''Requirements for Local Conference Control'', Internet Draft draft-ott-mmusic-mbus-req-00.txt, Work in Progress, June 1999.

[3] J. Ott, C. Perkins, and D. Kutscher, ''A Message Bus for Conferencing Systems'', Internet Draft draf-ott-mmusic-mbus-transport-00.txt, Work in Progress, June 1999.

[4] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, ''RTP: A Transport Protocol for Real-Time Applications,'' RFC 1889, January 1996.

## Appendix A: RTP specific commands

The following commands are used to provide information about an RTP[4] media source. Each source in media sessions is identified by its SSRC (not by the CNAME, since this would not be unique). Correlation to CNAMEs for cross-media references (eg: for lip-synchronization) has to be done by receiving entities.

rtp.ssrc (ssrc)
Sent to inform an entity of the SSRC it is to use for the remainder of the session.

rtp.source.exists (ssrc validityTime)
The rtp.source.exists command is sent by a media engine to assert that a particular source is present in a session. The validityTime parameter is the time for which that source should be considered valid, in seconds. If another rtp.source.exists command has not been received for that source within this time period, the source is implicitly timed out. The validityTime SHOULD be three times the RTCP reporting interval for that session.

rtp.source.remove (ssrc)
The rtp.source.remove command is used to indicate that a source has left the session.

rtp.source.name (ssrc name)
rtp.source.email (ssrc email)
rtp.source.phone (ssrc phone)
rtp.source.loc (ssrc loc)
rtp.source.tool (ssrc tool)
rtp.source.note (ssrc note)
rtp.source.cname (ssrc cname)

These commands are used to pass RTCP SDES information from a media engine to a user interface. If sent to a media engine, they have no effect unless the ssrc field is the SSRC of that engine, in which case the are used to change the SDES information being transmitted by that media engine.

rtp.source.reception (ssrc packetsRecv packetsLost packetsMisordered jitter validityTime)
This command is used to pass RTCP RR information from a media engine to a user interface. The total number of packets received, lost and misordered are sent, together with the network timing jitter in milliseconds and a validity time for this report in seconds.

rtp.source.packet.loss (dest_ssrc src_ssrc loss% validityTime)
Sent by a media engine to indicate the instantaneous packet loss

observed between two sources. The validityTime for this report is in milliseconds.

rtp.source.active (ssrc validityTime)
rtp.source.inactive (ssrc)

The rtp.source.active command indicates that a source is transmitting data into the session. The validityTime field indicates the period for which this source should be considered active, in milliseconds. The source.inactive command indicates that the source has stopped transmitting.

rtp.source.mute (ssrc muteState)
The rtp.source.mute command indicates that a source is to be muted/ unmuted. The value of the muteState parameter is 0 to indicate unmuted, and 1 to indicate muted.

rtp.source.packet.duration (ssrc packetDuration)
Sent by a media engine to indicate the duration, in milliseconds, of packets received from a source. This may be used to control the duration of packets sent by a media engine, if sent to that engine with the cname of the engine.

rtp.source.codec (ssrc codec)
Sent by a media engine to indicate the codec in use by a source.

rtp.source.playout (ssrc playoutDelay)
Sent by a media engine to indicate the playout delay, in milliseconds, for a source (that is, end-to-end time from capture to playout). This allows for lip-synchronization between audio and video streams.

## Appendix B: Media Engine Control Commands -- Audio

The following commands are used to control the playout of data by the media engine. If sent to a media engine they change output settings, if sent by a media engine they inform other entities of the state of the output.

It is expected that a similar set of commands will be defined to control the operation of other media engines (eg: video, shared-workspace).

audio.output.gain (gain)
: Select gain (volume) of the output device.

audio.output.port (port)
: Select port to be used. The allowable values for the port are ``speaker'', ``line'' and ``headset''.

audio.output.mute (muteState)
: Mute (if muteState is 1) or unute (if muteState is 0) the output.

audio.output.powermeter (level%)
: Sent from media engine to user interface to control display of audio power meters. Level is specified as a percentage.

audio.output.agc (state)
: Enable/disable automatic gain control for output. state is 1 to enable, 0 to disable.

audio.output.synchronize (state entity)
: Enable synchronization with the Mbus entity specified. This uses the rtp.source.playout messages to achieve synchronization (eg: lip-sync between audio and video). The value of state is 0 to disable synchronization, 1 to enable. Note that this option may conflict with the audio.output.playout.delay.* commands.

audio.input.gain (gain)
audio.input.port (port)
audio.input.mute (muteState)
audio.input.powermeter (level)
audio.input.agc (agcState)

: These commands function in a manner analogous to the audio.output.* commands, but affect input of data by the media engine.

audio.codec.describe (codec name clock-rate channels)
: Describe a codec. The codec parameter is an opaque codec identifier. The name parameter provides a human readable description of the codec, clock-rate and channels are self explanatory.

For an RTP-based media tool, the codec parameter will be an RTP payload type number.

audio.codec.supported.tx (codec ...)
audio.codec.supported.rx (codec ...)

Indicates that the specified codec(s) are supported by this media engine. This list includes basic codecs only, repair schemes such as redundancy, FEC and interleaving are not included in the response.

The audio.codec.supported.tx indicates codecs which this media engine can transmit. The audio.codec.supported.rx indicates codecs which are supported for reception.

audio.codec.supported.tx.request
audio.codec.supported.rx.request

Request that the recipient responds with a codec.supported message.

audio.codec (codec)
If sent to a media engine, select the codec to be used when transmitting. If sent by a media engine, indicates the codec being used when transmitting.

audio.suppress.silence (state)
Enable/disable silence suppression on the input signal. State is 1 to enable, 0 to disable.

audio.channel.coding (coding parameters ...)
Indicate the channel coding scheme to be used. The following values are currently defined for coding

none - No special channel coding.

redundant - RFC2198 redundancy

interleaved - Interleaving.

Others may be defined in future.

The parameters section is specific to a particular encoding. For redundant coding, parameters are a series of codec, offset pairs. For interleaved coding, the number of blocks and block separation are parameters.

audio.channel.repair (repair-scheme)
Indicate the loss-repair scheme to be used at the receiver. The following repair schemes are currently defined:

- repetition
- pattern-match

Others may be defined in future.

Appendix C: Security commands

security.encryption.algorithm (algorithm)
security.encryption.key (key)

Specify the encryption algorithm and key to be used. The algorithm defaults to ``DES'' if not specified.

security.encryption (state)
Enable/disable encryption. Enable if state is 1, disable if state is 0. A security.encryption.key command MUST be sent before this command.

## Appendix D: Tool configuration

IETF conferencing tools so far make use of a variety of to some degree standardized command line options, including (but not limited to) the following:

```
tool -C conference -N name ... mc-addr/port/ttl
```

In order to accommodate the aforementioned boot phase procedures, the following parameters with the associated behavior are needed:

-wait [condition] Wait for a particular event to happen; if no condition is specified on the command line, an internally pre-defined condition is used. The application entity is supposed to start emitting self-announcements to the Mbus but should defer any further action (with respect to self-configuration and media exchange) until the condition is met.

Besides waiting for the indicated condition to occur, an application entity is required to react to Mbus configuration requests directed to it, since their execution may be a prerequisite for it eventually receiving the ''GO'' message.

-nowait Start immediately.

Examples

rat — Start and go ahead with whatever you want to do.

rat -wait ... — Takes default condition: Wait for the own user interface to show up and configure the engine.

rat -wait controller — Wait for a local conference controller to show up and make the application continue.