Network Working Group
Internet-Draft
Expires: August 15, 2001

Kutscher
TZI, Universitaet Bremen
February 14, 2001

The Message Bus: Guidelines for Application Profile Writers

draft-ietf-mmusic-mbus-guidelines-00.txt

Status of this Memo

Copyright Notice

Abstract

This memo defines a list of conventions for terminology, algorithms and procedures for interaction models that are useful for applications using the Message Bus (Mbus) [1]. These conventions are intended as guidelines for designers of Mbus application profiles and Mbus implementations/applications.

Version Info

$Revision: 2.17 $

$Date: 2001/02/14 14:55:07 $

Table of Contents

1. Introduction

[1] specifies the Mbus transport protocol. This includes the basic protocol behaviour, representation of PDUs and PDU elements and operational aspects, such as Mbus configuration. This base specification can be used to implement the Mbus protocol. Specific Mbus command sets are not defined in this specification -- they are expected to be defined in application specific documents.

This document builds on the basic transport specification and tries to give useful recommendations for writers of Mbus application profiles in the following areas:

Terminology: We propose common Mbus related terms in order to unify the terminology used in Mbus application profiles.

Algorithms: A set of algorithms are given that are useful for Mbus implementors.

Notation Conventions: We propose a set of conventions that can be used for writing Mbus application profiles, such as recommendations how to specify the characteristics of an Mbus command etc.

Representation Conventions: Building upon the representation of values given in the Mbus transport specification we define additional representations for more complex data types.

Interaction Models: The transport specification essentially defines one interaction model for the Mbus: Message passing (with support for group communication). In this document we propose conventions for additional interaction models that can be implemented with the basic Mbus message passing mechanisms.

2. Terminology

This section defines some Mbus related terms.

Application profile: A specification of Mbus commands, their semantics and characteristics for a specific application context.

Fully qualified Mbus address: A unique, fully qualified Mbus address that denominates exactly one concrete existing Mbus entity at a given time and can thus not be expanded further.

Addressing scheme: A set of Mbus address key and possible address values. An Mbus application profile defintion SHOULD contain a specification of a corresponding addressing scheme.

3. General Mbus Command Definition Conventions

This section gives some general recommendations how to specify Mbus commands and their parameter lists and how to represent certain data types using Mbus parameter types.

3.1 Conventions for parameter encoding

This section list some useful conventions for encoding values of commonly used parameter types.

pair: A pair is a list that has exactly two elements, not necessarily of the same type.

map: A map is a list of pairs where the first element of all pairs (the keys) is of type T1 and the second element of all pairs (values) is of type T2. A key value may occur only once. T1 and T2 may be equal. The map is specified as "map of (T1,T2)".

MbusAddressElement: A Mbus address element is represented as a pair of strings: The first element represent the address element key and the second element represents the address element value.

MbusAddress: A Mbus address is represented as a list of MbusAddressElements, a map of (string,string).

3.2 Parameter Specification for Mbus Commands

This section lists some guidelines how parameters of Mbus commands should be specified in Mbus application profiles.

1. Each parameter SHOULD be associated with a well-defined Mbus parameter type, i.e., one of the types specified in [1].

2. Homogeneous lists (i.e. lists that contain elements that are all of the same type) SHOULD be declared by specifying the element type, for example "list of string".

3. For heterogeneous lists the type of each element SHOULD be specified.

4. In the case an optional parameter list is allowed, it SHOULD be the last parameter and a list of the potential parameters and their name tags SHOULD be given.

See Section 3.2.1 for recommendations how to specify and use optional parameters.

3.2.1 Optional parameters

Some Mbus command argument lists may be of variable length, i.e., some parameters may be optional. The following conventions are proposed for optional parameters:

- An argument list can have elements that are "optional argument lists". The elements of those lists are pairs: Each first element is of type String and represents a name for an optional parameter. Each second element is the value and can be of arbitrary type. This allows applications to process optional parameters independent of their order even when some parameters are missing.

- A command specification may have a set of predefined optional parameters. In a command definition, these SHOULD be declared by listing the names and type definitions. If applicable, naming conventions for further parameters SHOULD be specified, for example to distinguish different classes of optional parameters.

- If a command definition requires optional parameters it SHOULD provide exactly one optional parameters list. It is RECOMMENDED that the optional parameter list be the last element of the command's parameter list, as shown in Section 3.2.1.2.

3.2.1.1 Example of a Definition of an Optional Parameter List

```
tools.foo.bar
    Remote command (reliable/unreliable)

Parameters:
  p1: string
    p1 is the value for...

  p2: int
    p2 is the number of...

  p3: list of string
    a list of names for...

Optional Parameters:
  p4: string
   the optional name of...

  p5: string
   the optional number of...
```

3.2.1.2 Example for an Mbus Command with an Optional Parameter

```
Entity A: ---------

"mbus/1.0" 42 65454365 U (app:foo module:gui id:4711-1@192.168.1.1) \
           (app:foo module:engine) ()
tools.foo.bar ("gg" 17 ("a" "b") ("p5" 42))
```

4. Usage of Mbus Addresses

Mbus addresses are lists of arbitrary key/value pairs and every Mbus entity can choose its own Mbus address. Target Mbus addresses can be partly qualified to allow for group addressing or selecting receivers by certain application specific key elements that represent a certain application or service type. Except for the mandatory id-element the Mbus transport specification [1] does not define any other elements and it is suggested that Mbus application profile definitions specify a set of useful address element names and values for a specific application context.

Example of a fully qualified Mbus address and three partly qualified Mbus addresses:

```
(conf:test media:audio module:engine app:rat id:4711-1@192.168.1.1)
(media:audio module:engine)
(module:engine)
()
```

These address elements might be used to offer a particular service to other entities and to disambiguate entities sufficiently. Address elements might also be used to express membership of a certain Mbus address group. When it is known that an entity will always send certain messages to a specific address group, an entity will have to provide the corresponding address elements (with proper keys and values) to become a member of that group. This depicts the following uses of Mbus address elements:

1. to signal affiliation to an application context

2. to offer a certain service

3. to receive messages for a certain subgroup (to "tune" to a specific sub-channel on the Mbus)

It is possible that for a given application context not every address element is to be used by every involved Mbus entity. Instead some elements (or values) might be reserved for use by "service providing entities" while others might be required in order to receive messages that are addressed to a certain subgroup.

Moreover it should be noted that it may make sense for entities to adopt more than one command profile and thus make use of more than one addressing scheme. An entity could provide all address elements that are required by command profile A and additionally provide all the required element for profile B.

In the following a list of guidelines is presented how to specify an

Mbus addressing scheme for an Mbus profile definition.

A Mbus profile definition SHOULD be a sufficiently self-contained specification of Mbus commands for a particular application area together with a specification of an addressing model for Mbus entities that are supposed to implement or use the commands. It SHOULD be clear which commands belong to an application profile definition and what addressing conventions are to be considered.

The following specifies how addressing conventions SHOULD be defined:

1. List the address element keys that are used.

2. List (or describe) the set of legal values for each element key and explain the semantics (if applicable).

3. Name the services (or mandatory commands) that an entity providing certain address element keys/values must provide/understand.

4. Give examples for complete Mbus addresses using the defined address elements.

5. For each command that is specified in the profile definition explain the relation to the address elements. ("This command is sent to the group xy." or "This command must be understood by every entity that provides the address element xy:z.")

4.1 Example of a Specification of an Addressing Scheme

TBD

5. Mbus Command Classes

The general semantic model for Mbus commands is that commands are sent as payload of messages from one peer to another receiving (group of) peer(s) in order to trigger some kind of operation on the receiving side. On a low level of abstraction every Mbus command can be modeled like this. However on a higher level of abstraction some classes of commands can be identified that are used to implement specific Mbus communication scenarios. The following list decribes these command classes briefly:

Remote commands: Remote commands are used to trigger an asynchronous operation on the target system. The command has a name that is associated with a certain operation on the receiving side and can be sent together with a list of arguments (that can be empty) that are interpreted by the receiver. The name and the type definition of the command are specified in application semantics definition document. See Section 5.1 for a detailed discussion of generic remote commands.

RPC-style commands: RPC-commands allow to associate an operation at an remote entity with an Mbus command and SHOULD be used when a caller expects a result message from the callee that can return result parameters of the remote procedure/function call. Different types of RPC-commands are defined in this specification. See Section 5.2 for a detailed discussion of RPC-style commands.

Transactions: Transaction-style commands are similar to remote commands because they are also used to trigger a remote operation. Additionally transactions are used in scenarios where the caller is interested in how/whether the remote operation has been performed. In general, characteristics of transactions are atomicity (recoverability), consistency, isolation (serializability) and durability. This specification defines procedures for atomic transactions. Consistency, isolation and durability are to be provided by the Mbus application themselves. See Section 5.3 for a detailed discussion of transactions.

Properties: Obtaining the value of a named property of another Mbus entity is a variant of an RPC-style command: One command is sent that represent a query and one command is returned to the caller that contains the value. Setting the value of a named variable is a simple remote command with a parameter for the new value. See Section 5.4 for a detailed discussion of commands related to property inspection and manipulation.

Event notification: An entity that frequently sends messages to inform other entities of certain events sends a command for each

state change (or after a certain interval) to a (group of) receiver(s). These commands are similar to the simple remote commands because they are also sent asynchronously. See Section 5.5 for a detailed discussion of event notification.

Contexts: Instead of short time interactions between entities that can be accomplished by RPCs and other command classes contexts allow for more persistent relationships between entities. Contexts are scopes for coherent commands that are to be exchanged within a long-term interaction. Contexts provide the service of assigning a name to an interaction context that allows to disambiguate Mbus interactions that use the same commands but refer to different contexts at the same time. See Section 5.6 for a detailed discussion of contexts.

The following sections specify useful procedures that SHOULD be considered when defining Mbus command sets that contain commands of the aforementioned classes:

5.1 Remote Commands

Simple remote commands (that do not belong to any of the other classes) require no special procedures or conventions besides the general recommendations for Mbus command definitions: They SHOULD be defined in a self-contained profile definition, their applicability, the command name and the command arguments SHOULD be documented like proposed in Section 3.2.

5.1.1 Example for a Command Definition

```
tools.foo.bar
    Remote command (reliable/unreliable)

Parameters:
  p1: string
    p1 is the value for...

  p2: int
    p2 is the number of...

  p3: list of string
    a list of names for...

Optional parameters: none
```

5.1.2 Example for an Mbus Interaction

```
Entity A:
---------

"mbus/1.0" 42 65454365 U (app:foo module:gui id:4711-1@192.168.1.1) \
           (app:foo module:engine) ()
tools.foo.bar ("gg" 17 ("a" "b"))


Entity B:
---------

"mbus/1.0" 42 65454365 U  (app:foo module:engine id:4712-1@192.168.1.1) \
           (app:foo module:gui id:4711-1@192.168.1.1) ()
tools.foo.ok ()
```

5.2 RPC-style Commands

RPC-style commands are implemented by a command-pair: One command (with arguments) for triggering the remote procedure call and one command for the result. There are RPCs for Mbus "point-to-point" communication and RPC for Mbus group communications. The following conventions are proposed:

5.2.1 Invoking Remote Procedures

- No restriction is imposed on the command name. The class of the command ("RPC") SHOULD be mentioned in the command definition.

- The first argument of a RPC command SHOULD be a map of (string,string) (see Section 3.1) and contain meta information. The map SHOULD contain an element with the key "ID". The corresponding value is an arbitrary RPC ID that SHOULD be unique for the calling entity. For point-to-point RPCs (see Section 5.2.5 for RPCs to groups of entities) the meta information map SHOULD also contain an element with key "RPC-TYPE" and value "UNICAST". It is RECOMMENDED that unicast RPCs be sent using reliable Mbus messages. Multicast RPCs are defined in Section 5.2.5.

- The second argument of a RPC command is of type list and contains an arbitrary number of RPC parameters. The syntax and the semantics of this list SHOULD be defined in the definition of the RPC command.

5.2.2 Returning Results from Remote Procedure Calls

- The names of commands for returning RPC result are constructed using the name of the trigger command and appending the string ".return".

- The first argument of a RPC return command is a map of (string,string) for meta information that contains the following information:

  The original RPC ID that has been generated by the calling entity;

  Another element with the key of "RPC-STATUS" SHOULD have one of the following values:

  OK: The procedure has been called.

  UNKNOWN: No operation is associated with the RPC command. The command is unknown to the callee.

  The RPC-STATUS parameter is used to signal the generic RPC status and can be used to acknowledge the call of the specified RPC.

- The second argument of a RPC return command is of type list and contains a list of return parameters. It is RECOMMENDED that the first element of this list be of type list, containing application specific status information.

- The second element SHOULD be of type list and contain further application specific return parameters. The syntax and the semantics of this list SHOULD be defined in the definition of the RPC command.

- The application specific status information list SHOULD contain:

  An identifier (type symbol) that signals the general result (successful or unsuccessful operation) of the RPC. The possible values are "OK" and "FAILED".

  An identifier (type symbol) that represents the application specific result status of the procedure call. The set of symbols SHOULD be specified in the definition of the RPC.

  A textual description of the status (type string).

  If the general result of the RPC is "FAILED" the further parameters may be omitted although they have been specified in

the RPC definition.

5.2.3 Example for a Command Definition

The meta information list (for ID and RPC-TYPE) and the application status list does not have to be provided in RPC command definitions.

```
tools.foo.bar
    RPC

p1: string
    p1 is the value for...

p2: int
    p2 is the number of...

p3: list of string
    a list of names for...

optional parameters: none

The following application specific result states are defined:

BAR_COMPLETED     The bar operation has been called successfully.
NO_SUCH_P1        The p1 parameter is invalid.
FOO_CRASH         The foo module crashed during the execution of ba

return parameters:

r1: int
       the value of...
```

5.2.4 Example for an Mbus Interaction

```
Entity A:
---------

"mbus/1.0" 42 65454365 R (app:foo module:gui id:4711-1@192.168.1.1) \
           (app:foo module:engine id:4712-1@192.168.1.1) ()
tools.foo.bar ((("ID" "123") ("RPC-TYPE" "UNICAST")) \
           ("gg" 17 ("a" "b")))


Entity B:
---------

"mbus/1.0" 57 65454366 U  (app:foo module:engine id:4712-1@192.168.1.1) \
           (app:foo module:gui id:4711-1@192.168.1.1) ()
tools.foo.bar.return ((("ID" "123") ("RPC-STATUS" "OK")) \
           ((OK BAR_COMPLETED "Success!") (1)))
```

5.2.5 RPC Communication with Multiple Entities

Different scenarios for RPCs that are addressed to groups of entities are defined:

anycast
: A sender sends an RPC message to a group of entities and wants exactly one of the entities to perform the operation and return results.

one RPC, more than one result command
: A sender sends an RPC message to a group of entities and wants each entity to perform the operation and to return a result.

one RPC, coordinated result
: A sender sends a RPC message to a group of entities and expects each entity to perform the operation but only one result messages that represents all results of the addressed group.

5.2.5.1 Anycast

The following conventions are proposed for anycast RPCs:

The sender uses a group address as the Mbus target address of the RPC message using unreliable Mbus message transport.

The command meta-information list SHOULD provide an entry with key "RPC-TYPE" and value "ANYCAST".

Those of the receiving entities that want to respond to the RPC

and are able to perform the requested operation return a "standby" command in order to signal their disposition to provide the service. The name of the command is the RPC command name concatenated with ".standby". The first argument is again a meta-information list that contains the original RPC-ID. The target address of this command is an aggregate of the sender address and the target addres of the RPC und MUST therefore be sent unreliably. See Section 7.1 for a description of an address aggregation algorithm.

After a timeout T_anycast (Section 8) the entity that originally sent the RPC message selects one of the entities that answered with a "standby" command and sends it the RPC again (in a new message). This message MUST be sent using reliable Mbus message transport. The meta-information list of the command contains an additional entry with a key "REFERENCE". The value is the sequence number of the received standby message.

The entity that receives the second RPC message now operates as specified for the regular unicast RPC case.

5.2.5.1.1 Example for an Mbus Interaction

```
Entity A:
---------

"mbus/1.0" 42 65454365 U (app:foo module:gui id:4711-1@192.168.1.1) \
           (module:engine) ()
tools.foo.bar ((("ID" "123") ("RPC-TYPE" "ANYCAST")) ("gg" 17 ("a" "b")))


Entity B:
---------

"mbus/1.0" 57 65454366 U (app:foo module:engine id:4712-1@192.168.1.1) \
           () ()
tools.foo.bar.standby ((("ID" "123")))

Entity C:
---------

"mbus/1.0" 83 65454366 U (app:xy module:engine id:4713-1@192.168.1.1) \
           () ()
tools.foo.bar.standby ((("ID" "123")))


Entity A:
---------
```

```
"mbus/1.0" 43 65454367 U (app:foo module:gui id:4711-1@192.168.1.1) \
           (app:xy module:engine id:4713-1@192.168.1.1) ()
tools.foo.bar ((("ID" "123") ("RPC-TYPE" "ANYCAST") ("REFERENCE" 83)) \
           ("gg" 17 ("a" "b")))
```

Entity C:

```
"mbus/1.0" 84 65454368 U (app:xy module:engine id:4713-1@192.168.1.1) \
           (app:foo module:gui id:4711-1@192.168.1.1) ()
tools.foo.bar.return ((("ID" "123") ("RPC-STATUS" "OK")) \
                ((OK BAR_COMPLETED "Success!") (1)))
```

5.2.5.2 One RPC, more than one Result Command

The following conventions are proposed for RPCs that are sent to a group of entities where each entity responds independently:

The sender uses a group address as the Mbus target address of the RPC message.

The command meta-information list SHOULD provide an entry with key "RPC-TYPE" and value "MULTICAST".

The sending entity sends the RPC in a message addressed to an Mbus address group using unreliable Mbus message transport and calculates the set of real Mbus addresses (see Section 2) of the entities that are enclosed in the destination address group. See Section 7.2 for an algorithm for expanding Mbus group addresses to real Mbus addresses.

The receiving entities operate as specified for the regular unicast RPC case, i.e. they try perform the operation and report the results to the sender of the RPC. The destination address of the result message MUST be the address of the RPC's sender. The message MUST be sent reliably.

After an application dependent timeout the entity that originally sent the RPC evaluates the received results: If all entities return a RPC-STATUS of "OK" the RPC can be considered successful. The procedure of how return parameters are gathered, collapsed and presented to the user is application/implementation specific.

5.2.5.2.1 Example for an Mbus Interaction

```
Entity A:
---------

"mbus/1.0" 42 65454365 U (app:foo module:gui id:4711-1@192.168.1.1) \
          (module:engine) ()
tools.foo.bar ((("ID" "123") ("RPC-TYPE" "MULTICAST")) ("gg" 17 ("a" "b")))


Entity B:
---------

"mbus/1.0" 57 65454366 U  (app:foo module:engine id:4712-1@192.168.1.1) \
          (app:foo module:gui id:4711-1@192.168.1.1) ()
tools.foo.bar.return ((("ID" "123") ("RPC-STATUS" "OK")) \
          ((OK BAR_COMPLETED "Success!") (1)))


Entity C:
---------

"mbus/1.0" 83 65454366 U (app:xy module:engine id:4713-1@192.168.1.1) \
          (app:foo module:gui id:4711-1@192.168.1.1) ()
tools.foo.bar.return ((("ID" "123") ("RPC-STATUS" "OK")) \
          ((OK BAR_COMPLETED "Success!") (2)))
```

5.2.5.3 one RPC, coordinated result

The following conventions are proposed for RPCs that are sent to a group of entities where each entity performs the operation but only one result messages that represents all results of the addressed group is sent to the original sender of the RPC:

The sender uses a group address as the Mbus target address of the RPC message.

The command meta-information list SHOULD provide an entry with key "RPC-TYPE" and value "COORDINATED".

The sending entity sends the RPC in a message addressed to an Mbus address group using unreliable Mbus message transport.

The receiving entities try to perform the operation and then send intermediate result commands to the RPC destination group. After a timeout T_Coordination one entity aggregates all intermediate results and sends an aggregated RPC-result message to the original sender. The coordination process and the procedure how to decide which entity reports the gathered results is yet TBD.

:-(

5.2.5.3.1 Example for an Mbus Interaction

Entity A:

```
"mbus/1.0" 42 65454365 U (app:foo module:gui id:4711-1@192.168.1.1) \
           (module:engine) ()
tools.foo.bar ((("ID" "123") ("RPC-TYPE" "COORDINATED")) ("gg" 17 ("a" "b")))
```

TBD...

5.3 Transactions

Transactions are implemented by defining a command that triggers an operation and an additional acknowledgement command that is sent after the operation has completed (or failed). Acknowledgement commands MUST refer to the initial trigger command and this relation is expressed by a special reference parameter that is generated by the caller. Note that the acknowledgement command is different from acknowledgments on the Mbus transport level: Those only ensure that messages are really received by the addressees, whereas transaction acknowledgments inform the original caller about the result of a certain operation that the callee should have performed upon reception of the transaction command.

Transaction commands are only allowed for unicast messages, they may not be sent to address groups. They MUST be sent using reliable Mbus messages. Senders of transaction commands are called clients, receivers of transaction commands are called servers.

After a sender (a client) has initiated a transaction with the respective transaction command (see below) it may abort (rollback) the transaction with a dedicated command (see below) or finally commit the transaction using another dedicated command.

It should be noted that means for concurrency control, e.g., to achieve consistency in the presence of parallel transactions, have to be provided by the application itself and is not part of these conventions.

The following conventions for transaction commands are proposed:

- There are no restrictions on naming transaction commands. Any command in an Mbus command hierarchy can be used for triggering transactions. The class of the command ("TRANSACTION") SHOULD be mentioned in the command definition.

- The first argument of a transaction command SHOULD be a map of (string,string) (see Section 3.1) and contain meta information. The map SHOULD contain an element with the key "ID". The corresponding value is an arbitrary transaction ID that SHOULD be unique for the calling entity.

- The second argument of a transaction command is of type list and contains an arbitrary number of transaction parameters. The syntax and the semantics of this list SHOULD be defined in the definition of the transaction command.

- After a transaction command has been sent the sender can either cancel or commit the transaction: A transaction cancel command has the original transaction command name plus an appended ".cancel" as a command name and the original transaction id as a meta information parameter. A receiver SHOULD cancel or rollback any actions initiated by the original transaction message after receiving a transaction cancellation and delete any state related to the transaction. A transaction commit command has the original transaction command name plus an appended ".commit" as a command name and the original transaction id as a meta information parameter. A receicing entity SHOULD finish outstanding action initiated by the original transaction command after receiving a transaction commit command and delete any state related to the transaction. After a commit has been received cancel commands for the corresponding transaction MUST NOT be honoured anymore.

- After receiving a transaction command an entity responds with an acknowledgement. Acknowledgment command names are constructed using the name of the trigger command and appending the string ".ack". The first argument of a transaction acknowledgement command is of type "String" and contains the original transaction ID that has been generated by the calling entity. Any action that is performed MUST be reversible and SHOULD only be executed in non-reversible way after a commit command has been received for the corresponding transaction. If a cancel command for the corresponding transaction has been received before a commit command the entity SHOULD rollback any performed actions.

- The second argument of a transaction acknowledgement command is of type "Symbol" and can have one of the following values:

  OK: The transaction was successful.

  UNKNOWN: No operation is associated with the transaction command. The command is unknown to the callee.

  FAILED: The transaction could not be performed successfully.

CANCELLED: The transaction has been cancelled.

- Further information about the transaction status can be supplied optionally and can be passed in a common optional command list (see Section 3.2.1).

- After a server has received a commit command for a transaction it SHOULD respond with an additional acknowledgement command. For clarity the command name for this command is composed of the name of the original command and an appended ".completed". The parameters are the same as for the first acknowledgement.

5.3.1 Example for a Command Definition

The meta information list (for the transaction ID) does not have to be provided in transaction command definitions.

```
tools.foo.bar
    TRANSACTION

Parameters:

  p1: string
    p1 is the value for...

  p2: int
    p2 is the number of...

  p3: list of string
    a list of names for...

Optional parameters: none
```

5.3.2 Example for an Mbus Interaction I

```
   Entity A:
   ---------

   "mbus/1.0" 42 65454365 R (app:foo module:gui id:4711-1@192.168.1.1) \
              (app:foo module:engine id:4712-1@192.168.1.1) ()
   tools.foo.bar ((("ID" "123")) ("gg" 17 ("a" "b")))


   Entity B:
   ---------

   "mbus/1.0" 57 65454366 U  (app:foo module:engine id:4712-1@192.168.1.1) \
              (app:foo module:gui id:4711-1@192.168.1.1) ()
   tools.foo.bar.ack ("123" OK)


   Entity A:
   ---------

   "mbus/1.0" 43 65454367 R (app:foo module:gui id:4711-1@192.168.1.1) \
              (app:foo module:engine id:4712-1@192.168.1.1) ()
   tools.foo.bar.commit ((("ID" "123")))


   Entity B:
   ---------

   "mbus/1.0" 58 65454368 U  (app:foo module:engine id:4712-1@192.168.1.1) \
              (app:foo module:gui id:4711-1@192.168.1.1) ()
   tools.foo.bar.completed ("123" OK)
```

5.3.3 Example for an Mbus Interaction II

```
Entity A:
---------

"mbus/1.0" 42 65454365 R (app:foo module:gui id:4711-1@192.168.1.1) \
           (app:foo module:engine id:4712-1@192.168.1.1) ()
tools.foo.bar ((("ID" "123")) ("gg" 17 ("a" "b")))


Entity B:
---------

"mbus/1.0" 57 65454366 U  (app:foo module:engine id:4712-1@192.168.1.1) \
           (app:foo module:gui id:4711-1@192.168.1.1) ()
tools.foo.bar.ack ("123" OK)


Entity A:
---------

"mbus/1.0" 43 65454367 R (app:foo module:gui id:4711-1@192.168.1.1) \
           (app:foo module:engine id:4712-1@192.168.1.1) ()
tools.foo.bar.cancel ((("ID" "123")))


Entity B:
---------

"mbus/1.0" 58 65454368 U  (app:foo module:engine id:4712-1@192.168.1.1) \
           (app:foo module:gui id:4711-1@192.168.1.1) ()
tools.foo.bar.completed ("123" CANCELLED)
```

5.4 Inspection/Modification of Properties

Obtaining the value of a named property of another entity is achieved by using a set of RPC-style commands (see Section 5.2): RPCs are defined for setting, obtaining and "watching" values of properties. The following conventions are proposed:

- No restriction is imposed on the property's name. Entities can use any command to transmit a new value for a certain property. The Mbus command name is the property name. In a profile definition a property SHOULD be classified as ("PROPERTY") which means that it is possible for other entities to set/retrieve its value (see below). Additionally the type of the property SHOULD be specified using the syntax specified in Section 3.2.

- In order to obtain the value of a certain property that is

managed by another Mbus entity a module sends a "get-request" RPC to the respective entity. The command name of this RPC is composed of the property name and ".get". The RPC argument list is empty. Upon receiving a get-request an entity hosting the property returns a RPC result command to the requesting entity. The (only) RPC argument is the current value. Application specific status values (Section 5.2) for the return command of a get-RPC are:

OK: Property exists.

NO_SUCH_PROPERTY: The requested property does not exist.

- In order to change the value of a certain property that is managed by another Mbus entity a module sends a "set-request" RPC to the respective entity. The command name of this RPC is composed of the property name and ".set". The (only) RPC argument is the new value. Upon receiving a set-request an entity hosting the property changes the value and returns a RPC result command to the requesting entity. The (only) RPC argument is the new value. Application specific status values (Section 5.2) for the return command of a set-RPC are:

  OK: Property exists and has been updated.

  NO_SUCH_PROPERTY: The requested property does not exist.

- Besides get-requests clients can also use "watch-requests" to obtain the value of properties. watch-requests can be sent if an entity wants to be informed about any updates to the property value. The RPC command name of this request is composed of the variable name and ".watch". The RPC argument list is empty. Upon receiving a watch-request an entity that hosts the property adds the originating entity to a list of subscribers for the property variable and will further on send an update to all list members when the value of the property changes. The current value of the property is sent to the originator of the watch-request in a RPC return command. The updates are event notificatons as specified in Section 5.5, i.e., simple Mbus commands with the property name as the command name and the current value as the only parameter. Application specific status values (Section 5.2) for the return command of a watch-RPC are:

  OK: Property exists and the requesting entity has been added to the list of subscribers.

  NO_SUCH_PROPERTY: The requested property does not exist.

- A subscriber of a property can also send an "unwatch-request" RPC

to unsubscribe. The command name of this request is composed of the property name and ".unwatch". The argument list is empty. The RPC return command also requires no further RPC parameter. Application specific status values (Section 5.2) for the return command of a uwatch-RPC are:

OK: Property exists and the requesting entity has been removed from the list of subscribers.

NO_SUCH_PROPERTY: The requested property does not exist.

NOT_SUBSCRIBED The requesting entity is not on the list of subcribers for this property.

Entities that have been declared to provide a property, e.g., in a profile definition, SHOULD support all aforementioned requests.

All requests related to properties MUST be send as unicast RPCs.

Notes:
Requests for non-existing properties result in a RPC-UNKNOWN error (see Section 5.2).

5.4.1 Example for a Command Definition

The RPC commands for the different property request do not have to be specified.

```
tools.foo.bar
    PROPERTY

    type: string
```

5.4.2 Example for an Mbus Interaction I

```
   Entity A:
   ---------

   "mbus/1.0" 42 65454365 R (app:foo module:gui id:4711-1@192.168.1.1) \
              (app:foo module:engine id:4712-1@192.168.1.1) ()
   tools.foo.bar.get ((("ID" "123") ("RPC-TYPE" "UNICAST")))


   Entity B:
   ---------

   "mbus/1.0" 57 65454366 U  (app:foo module:engine id:4712-1@192.168.1.1) \
              (app:foo module:gui id:4711-1@192.168.1.1) ()
   tools.foo.bar.get.return ((("ID" "123") ("RPC-STATUS" "OK")) \
              ((OK OK "") ("the value")))
```

5.4.3 Example for an Mbus Interaction II

```
   Entity A:
   ---------

   "mbus/1.0" 42 65454365 R (app:foo module:gui id:4711-1@192.168.1.1) \
              (app:foo module:engine id:4712-1@192.168.1.1) ()
   tools.foo.bar.set ((("ID" "123") ("RPC-TYPE" "UNICAST")) \
              ((OK OK "") ("the value")))

   Entity B:
   ---------

   "mbus/1.0" 57 65454366 U  (app:foo module:engine id:4712-1@192.168.1.1) \
              (app:foo module:gui id:4711-1@192.168.1.1) ()
   tools.foo.bar.set.return ((("ID" "123") ("RPC-STATUS" "OK")) \
              ((OK OK "") ("new value")))
```

5.4.4 Example for an Mbus Interaction III

```
   Entity A:
   ---------

   "mbus/1.0" 42 65454365 R (app:foo module:gui id:4711-1@192.168.1.1) \
              (app:foo module:engine id:4712-1@192.168.1.1) ()
   tools.foo.bar.watch ((("ID" "123") ("RPC-TYPE" "UNICAST")))


   Entity B:
   ---------
```

```
"mbus/1.0" 57 65454366 R  (app:foo module:engine id:4712-1@192.168.1.1) \
          (app:foo module:gui id:4711-1@192.168.1.1) ()
tools.foo.bar.watch.return ((("ID" "123") ("RPC-STATUS" "OK")) \
          ((OK OK "") ("the value")))
```

Entity C:

```
"mbus/1.0" 82 65454367 R (app:bar module:engine id:4713-1@192.168.1.1) \
          (app:foo module:engine id:4712-1@192.168.1.1) ()
tools.foo.bar.set ((("ID" "345") ("RPC-TYPE" "UNICAST")) \
          ((OK OK "") ("new value")))
```

Entity B:

```
"mbus/1.0" 58 65454368 R  (app:foo module:engine id:4712-1@192.168.1.1) \
          (app:bar module:engine id:4713-1@192.168.1.1) ()
tools.foo.bar.set.return ((("ID" "345") ("RPC-STATUS" "OK")) \
          ((OK OK "") ("new value")))
```

Entity B:

```
"mbus/1.0" 59 65454369 U  (app:foo module:engine id:4712-1@192.168.1.1) \
          (app:foo module:gui id:4711-1@192.168.1.1) ()
tools.foo.bar ("new value")
```

Entity A:

```
"mbus/1.0" 43 65454370 R (app:foo module:gui id:4711-1@192.168.1.1) \
          (app:foo module:engine id:4712-1@192.168.1.1) ()
tools.foo.bar.unwatch ((("ID" "124") ("RPC-TYPE" "UNICAST")))
```

Entity B:

```
"mbus/1.0" 60 65454371 R  (app:foo module:engine id:4712-1@192.168.1.1) \
          (app:foo module:gui id:4711-1@192.168.1.1) ()
tools.foo.bar.unwatch.return ((("ID" "123") ("RPC-STATUS" "OK")))
```

5.5 Event Notifications

There are different usage scenarios for events that origin at a certain entity and need to be signaled to other entities. An event notification is an Mbus command with an arbitrary argument list that is sent (eventually, maybe periodically) to a group of entities. The following conventions are proposed:

- No restriction is imposed on the name of the notification command. In a profile definition a command SHOULD be classified as ("EVENT NOTIFICATION").

- A command that is classified as an event notification SHOULD also be associated with a default target address that is used when the notification command is sent.

See Section 6 for conventions of how to subscribe to and how to redirect event notifications.

5.5.1 Example for a Command Definition

```
tools.foo.bar
    EVENT NOTIFICATION

    default target address: (app:xy module:gui)

Parameters:
  p1: string
    p1 is the value for...

  p2: int
    p2 is the number of...

  p3: list of string
    a list of names for...
```

5.5.2 Example for an Mbus Interaction

Entity A:

```
"mbus/1.0" 42 65454365 U (app:foo module:gui id:4711-1@192.168.1.1) \
           (app:xy module:gui) ()
tools.foo.bar ("gg" 17 ("a" "b"))
```

5.6 Contexts

RPCs can be used to trigger a remote operation with the possibility to obtain results from a single operation thus representing a short time interaction between two or more Mbus entities. Sometimes there is the need for more persistent interaction relations between entities, for example, when a series of commands all refer to the same context. The command category "contexts" allows for expressing a long-term relationship between commands that are exchanged by Mbus entities.

The model that is used here is the concept of a specific context in which a sequence of Mbus commands are exchanged that relate to each other and are originated by entities of a group of Mbus entities. The context that provides a scope for Mbus commands is identified by a unique id that is used in all commands belonging to the context.

Contexts can start to exist (they can be created) and cease to exist (destructed). Mbus commands for context creation and destruction will be defined below.

Only certain specified commands are valid within a context. An Mbus context definition specifies these commands and their semantics. Context commands are either RPC commands (see Section 5.2) or event notifications that provide the context-id in the meta information parameter (key="CONTEXT-ID"). The subsequent argument of a context command is of type list and contains an arbitrary number of parameters. The syntax and the semantics of this list SHOULD be specified in the definition of the command.

The name of a context is an Mbus command prefix. Command names for construction and destruction commands as well as other context commands are derived from the context name by appending a dot and the name of the method. The name of the construction command is CONTEXT_NAME.create and the name of the destruction command is CONTEXT_NAME.delete. Both are "simple" Mbus commands (remote commands) with one parameter: the context-id as a parameter of type string.

CONTEXT_NAME.create and CONTEXT_NAME.delete can be sent to single Mbus entities as well as to group of entities using a Mbus group address. It is RECOMMENDED that context-creation/deletion messages to single entities be sent reliable. Only the creator of a context (the entity that has sent the CONTEXT_NAME.create message) SHOULD delete the corresponding context.

5.6.1 Example for a Command Definition

The meta information list (for the context ID and eventual RPC or

transaction IDs) does not have to be provided in each command definition in the context definition.

Context "conf.call-ctrl.call"

About: This context definition comprehends commands that can be use for a "call context". The following commands may refer to different contexts that represent different calls.

The following commands are defined in this context:

conf.call-ctrl.call.setup
RPC

Parameters:

caller: string
identifies the caller

callee: string
identifies the callee

Optional Parameters: none

conf.call-ctrl.call.disconnected
EVENT NOTIFICATION

default target address: (app:controller)

Parameters:

reason: string
the reason for the disconnection

Optional Parameters: none

5.6.2 Example for an Mbus Interaction

```
Entity A:
---------

"mbus/1.0" 42 65454365 U (app:controller module:engine id:4711-1@192.168.1.1)
           (app:call-ctrl module:engine id:4712-1@192.168.1.1) ()
conf.call-ctrl.call.create ("345")


Entity A:
---------

"mbus/1.0" 43 65454366 R (app:controller module:gui id:4711-1@192.168.1.1) \
           (app:call-ctrl module:engine id:4712-1@192.168.1.1) ()
conf.call-ctrl.call.setup ((("ID" "123") ("CONTEXT-ID" "345")) \
              ("joe@foo.bar" "bob@foo.bar"))


Entity B:
---------

"mbus/1.0" 57 65454380 U  (app:call-ctrl module:engine id:4712-1@192.168.1.1)
            (app:controller) ()
conf.call-ctrl.call.disconnected ((("CONTEXT-ID" "345")) ("hangup"))


Entity A:
---------

"mbus/1.0" 44 65454385 U (app:controller module:engine id:4711-1@192.168.1.1)
           (app:call-ctrl module:engine id:4712-1@192.168.1.1) ()
conf.call-ctrl.call.delete ("345")
```

5.7 Status Signaling

In order to notify other entities of status events asynchronously, each Mbus entity SHOULD send such events using the "mbus.status" command. This command is an event notification as specified in Section 5.5 and can thus also be given a default target address.

As specified in Section 5.5 the default target address of this message can be redirected using the "mbus.register" command defined in Section 6.1.1.

```
            mbus.status
                EVENT NOTIFICATION

          Parameters:
            class: symbol
                An identifier for the class of the status message.
                Predefined identifiers are:

                  INFO:    for informational messages
                  WARNING: for warnings
                  ERROR:   for error reports

                Application profiles can also define new status
                message classes.

            sym: symbol
                An identifier for the status message. Application
                profile definitions SHOULD enumerate the possible
                status symbols (and their textual description, see
                below).

            descr: string
                A textual description for the status message.

          Optional Parameters: none
```

To be discussed (FIXME): Should mbus.status only be used top-level or with arbitrary prefixes?

5.7.1 Example of a Definition of Status Symbols

TBD

6. Mbus service models

In general, the Mbus is a communication channel for message passing within a group of modules. Mbus implementations provide mechanisms to enable applications modules to pass messages to other modules. From an application point of view the goal of using the Mbus is using certain services of other entities (or providing these services).

Each Mbus entity can provide a number of different services: It may perform certain operations for other entities that are triggered by the reception of Mbus commands or it may notify one or more entities of events.

In the simplest case, an entity will simply receive Mbus messages and perform the operations that are denominated by the commands that are contained in the messages. Sometimes, however, entities will only process certain commands if they are are received from an entity that has registered as a client, e.g. a controller, before. Entities that are remote-controlled via their Mbus interface could restrict the number of controlling entities to one (at a time) in order to ensure consistency. Also, there could be event notifications that are sent to a certain dedicated controller only, as well as there can be notifications that can be sent to a group of receivers, each of which having subscribed to this event source before. Again, in simple scenarios, entities may just broadcast all event notifications to the whole Mbus.

It is proposed that all commands, variables, event notifications that an entity may send or receive in a specific application context be subsumed in a common Mbus command set definition. Service models that apply to such a set of Mbus commands SHOULD be specified as well.

In the following the different service models (control relation classes) are described in detail and a list of conventions and recommendations is presented that SHOULD be considered when writing Mbus command definitions.

Different classes of control relations are defined:

- no control
- tight control
- exclusive tight control

These different classes of control relations are usually applied to a command set that is implemented by some Mbus entities. It is

suggested that a control relation type is assigned to command sets in the command set definition.

The motivation for defining different service models is to accommodate different applications with different requirements concerning flexibility and the level of control for their Mbus communication.

The next sections specify the semantics and implications of the individual control classes:

6.1 No Control

"no control" means that entities do not require a special control relation with another entity to be established in order to accept commands from it. All Mbus commands, variables etc. of the respective command set can be used directly and there is no regulation of the number of entities that may interact using the respective commands at a time.

A command set that is classified as "no control" may contain commands for unsolicited event notification or even RPC-style commands that can be sent by an entity conforming to a specific Mbus command set definition. These Mbus commands that are originated by a conforming entity may be addressed to a default target address. There may be a default target address for all commands of a command definition set but each command may be associated to a specific default target address. It is RECOMMENDED that commands of the "no control" class that may be sent without prior solicitation, such as event notifications, are assigned a default target address.

The default target address that an entity sends unsolicited commands to may be changed by other entities. Entities may add themselves to a list of clients/controllers that is maintained by another service providing entity. The effect of having the service providing entity add another entity to a list of clients is that the default target address is no longer used but the respective messages are directed to the client entity. If more than one entity tries to add itself to the target address list it is up to the application to allow or deny this. Generally, entities of the "no control" class are expected to accept multiple clients. When multiple clients are present each message that would otherwise just be sent to the default target address is sent either to a Mbus group address that uniquely represents the registered clients or is sent independently to all clients. Clients may also deregister. When all clients have deregistered the entity SHOULD use the default target address for the respective command again.

The changing of the default target address is called redirection.

Redirection may take place on single commands or a complete command set. If a command or a command set use a default target address that can be redirected by clients it SHOULD be marked as "REDIRECTABLE" and the default target address SHOULD be given.

Redirection commands belong to the class of RPC-commands. The following commands are defined (see Section 3.1 for parameter type definitions):

6.1.1 mbus.register

RPC

Parameters:

command: string
Name of the Mbus command (or command set prefix, MUST be specified absolutely)

addr: MbusAddress
Mbus address that should be registered.

Optional Parameters: none

Description: This command is sent by an interested client entity to a service providing entity in order to change its default target address for the given command (prefix).

Application specific return values

OK: The client has been added to the address list.

NO_SUCH_COMMAND: The command (prefix) that has been given in the request is unknown.

DENIED: The requesting entity is denied to register the given command (prefix).

Return parameters:

addr-list: list of MbusAddress
the new list of registered clients.

6.1.2 mbus.deregister

RPC

Parameters:

command: string
Name of the Mbus command (or command set prefix, MUST be specified absolutely)

addr: MbusAddress
Mbus address that should be deregistered.

Optional Parameters: none

Description: This command is sent by a registered client entity to a service providing entity in order to deregister from a command or service subscription.

Application specific return values

OK: The client has been removed from the address list.

NO_SUCH_COMMAND: The command (prefix) that has been given in the request is unknown.

NOT_REGISTERED: The given address has not been registered before for the command (prefix).

Return parameters:

add-list: list of MbusAddress
the new list of registered clients.

6.1.3 mbus.registered

EVENT NOTIFICATION

Parameters:

string command: Name of the Mbus command (or command set prefix)

list of MbusAddress add-list: Current list of registered clients

Description: This notification is sent by a service providing entity after a new client has registered for a command (prefix). The second parameter contains the new list of registered clients for the given command. The notfication SHOULD be sent to the old list of clients (or to the default target address if no other clients have registered before).

6.1.4 mbus.get-registered

RPC

Parameters:

command: string
Name of the Mbus command (or command set prefix)

Optional Parameters: none

Description: This command can be used in order to obtain the current list of registered clients for the specified command (prefix).

Application specific return values

OK: The list of registered clients is provided.

NO_SUCH_COMMAND: The command (prefix) that has been given in the request is unknown.

Return parameters:

addr-list: list of MbusAddress
the list of registered clients.

6.2 Tight Control

An entity that requires "tight control" for some or all of its Mbus controllable resources will only accept commands from an entity that has established a control relationship before. This means that Mbus commands, variables etc. can only be used by another entity after it has registered itself as a "controller" at the entity that provides the resources. Upon this registration the controlled entity adds the new controller's Mbus address to a controller-address-list that is used for authorization and for sending event notifications etc. Another complementary de-registration command enables entitites to end the control relationship. Again, there is no regulation of the number of entities that may register themselves as a controller at a time.

Entities that conform to a command set definition marked as "tight control" SHOULD not send commands or event notifications to a default target address for resources of that set.

6.3 Exclusive Tight Control

"Exclusive tight control" has the same semantics as "tight control", except for the number of controllers at a time: An entity that

provides an Mbus command set that has been marked as requiring exclusive tight control will only accept one controller at a timer and reject register requests once a control relation with another entity has been established.

When a register request is received while another entity is currently registered as a controller the receicing entity SHOULD return the value "DENIED" (see Section 6.1.1.

7. Algorithms

7.1 Aggregation of Mbus Addresses

The following algorithm can be used to aggregate an arbitrary number of Mbus addresses: (FIX this example code)

```
template <class Input>
inline MAddress aggregate(Input start, Input end)
{
  typedef map<MAddressElement,int> elements;
  elements ae;
  int count=0;
  MAddress res;

// get all address elements:
  for(;start!=end;start++) {
    count++;
    for(MAddress::Const_Iterator ai(*start); ai; ++ai) {
      ae[*ai]++; // count occurence of AddressElements
    }
  }
  // keep all Elements that occured in every Address:
  elements::const_iterator ei;
  for(ei=ae.begin();ei!=ae.end();ei++) {
    if(ei->second==count) {
      res.setElement(ei->first.key(),ei->first.val());
    }
  }
  return res;
}
```

7.2 Expansion of Mbus Group Addresses

The following algorithm can be used to expand an Mbus group address to the set of real Mbus addresses enclosed within the group address.

TBD

8. Definition of Constants

The following constants are defined:

T_anycast: N_r * T_r (see [1])

T_Coordindation: 2 * N_r * T_r (see [1])

References

[1] Ott, J., Perkins, C. and D. Kutscher, "A Message Bus for Local Coordination", Internet Draft draft-ietf-mmusic-mbus-transport-04.txt, February 2001.

[2] Schulzrinne, H., Casner, S., Frederick, R. and V. Jacobsen, "RTP: A Transport Protocol for Real-Time Applications", RFC 1889, January 1996.

Author's Address

Dirk Kutscher
TZI, Universitaet Bremen
Bibliothekstr. 1
Bremen 28359
Germany

Phone: +49.421.218-7595
Fax: +49.421.218-7000
EMail: dku@tzi.org

Appendix A. Examples for Application Profiles

A.1 Mbus Profile for RTP applications

This needs to be updated.

The following commands are used to provide information about an RTP [2] 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.

The purpose of this Mbus profile is to provide a mechanism that allows for controlling RTP engine. RTP engines are entities that use an RTP protocol stack to send and receive RTP/RTCP data. This Mbus profile provide control commands to configure RTP engine and notification commands to notify interested engine of RTP events.

All commands of this set conform to the control class "exclusive tight control". The default destination address for event notifications is ().

It is suggested that RTP engines that support these commands, i.e. that can be controlled by the RPCs listed below and that can generate the event notifications, provide the following address element in their Mbus addresses:
(module:engine)

For all commands, event notifications that carry a SSRC value, the value is represented as a string in hexadecimal notation.

A.1.1 Configuring a RTP engine

The following commands are used to configure a RTP engine.

A.1.1.1 rtp.set-attributes

rtp.set-attributes (attribute-list)
RPC

This command is used to configure the SSRC and SDES parameters of a RTP engine.

Parameters:

attribute-list: map of (String,String)
A map containing configuration information. The first element of each pair is the name of the attribute and the second element of each pair is the value of the attribute. The following attribute

names are defined:

SSRC: The SSRC value to be used by the RTP engine.

NAME: The name of the participant.

PHONE: The phone number of the participant.

LOC: The geographic location of the participant.

TOOL: The application/tool name of the participant.

NOTE: The notice/status item.

CNAME: The canonical end-point identifier for the participant.

If other attribute names than those listed are used they are to be interpreted as PRIV SDES items (see [2]).

A.1.1.2 Controlling a RTP engine

The following commands are used to control a RTP engine during a RTP session.

A.1.1.2.1 rtp.source.mute

rtp.source.mute (ssrc muteState)
RPC

The command indicates that a source is to be muted/ unmuted.

Parameters:

ssrc: string
The SSRC value of the participant to be muted/unmuted.

muteState: Integer
The value of the muteState parameter is 0 to indicate unmuted, and 1 to indicate muted.

A.1.1.3 Events generated by a RTP engine

The following commands are used by a RTP engine to signal source specific events during a RTP session.

A.1.1.3.1 rtp.source.exists

rtp.source.exists (ssrc validityTime)
EVENT NOTIFICATION

The rtp.source.exists command is sent by a media engine to assert that a particular source is present in a session.

Parameters:

ssrc: string
The ssrc of the source this event notification refers to.

validityTime: Integer
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.

A.1.1.3.2 rtp.source.left

rtp.source.left (ssrc)
EVENT NOTIFICATION

The rtp.source.left command is used to indicate that a source has left the session.

Parameters:

ssrc: string
The ssrc of the source this event notification refers to.

A.1.1.3.3 rtp.source.attributes

rtp.source.attributes (ssrc attribute-list)
EVENT NOTIFICATION

This event notification is used to pass RTCP SDES information of other sources from a media engine to a user interface.

Parameters:

ssrc: string
The ssrc of the source this event notification refers to.

attribute-list: map of (String,String)
A map containing the SDES information. The first element of each pair is the name of the attribute and the second element of each pair is the value of the attribute. The same attributes as for rtp.set-attributes (Appendix A.1.1.1) are defined (except for SSRC).

A.1.1.3.4 rtp.source.reception

rtp.source.reception (ssrc packetsRecv packetsLost packetsMisordered jitter validityTime)
EVENT NOTIFICATION

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.

Parameters:

ssrc: string
The ssrc of the source this event notification refers to.

packetsRecv: Integer
Total number of received packets.

packetsLost: Integer
Total number of lost packets.

packetsMisordered: Integer
Total number of misordered packets.

jitter: Integer
Observed jitter in milliseconds.

validityTime: Integer
Validity time in seconds for this report.

A.1.1.3.5 rtp.source.packet.loss

rtp.source.packet.loss (dest_ssrc src_ssrc% validityTime)
EVENT NOTIFICATION

Sent by a media engine to indicate the instantaneous packet loss observed between two sources. The validityTime for this report is in milliseconds.

Parameters:

dest_ssrc: string
The ssrc of the receiving participant.

src_ssrc: string
The ssrc of the sending participant.

validityTime: Integer

The validityTime for this report is in milliseconds.

A.1.1.3.6 rtp.source.active

rtp.source.active (ssrc validityTime)
EVENT NOTIFICATION

The rtp.source.active notification 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.

Parameters:

ssrc: string
The ssrc of the source this event notification refers to.

validityTime: Integer
The validityTime field indicates the period for which this source should be considered active, in milliseconds.

A.1.1.3.7 rtp.source.inactive

rtp.source.inactive (ssrc)
EVENT NOTIFICATION

The rtp.source.active notifications indicates that a source has stopped transmitting data into the session.

Parameters:

ssrc: string
The ssrc of the source this event notification refers to.

A.1.1.3.8 rtp.source.packet.duration

rtp.source.packet.duration (ssrc packetDuration)
EVENT NOTIFICATION

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.

Parameters:

ssrc: string
The ssrc of the source this event notification refers to.

packetDuration: Integer
The duration, in milliseconds, of packets received from a source.

A.1.1.3.9 rtp.source.codec

rtp.source.codec (ssrc codec)
EVENT NOTIFICATION

Sent by a media engine to indicate the codec in use by a source.

Parameters:

ssrc: string
The ssrc of the source this event notification refers to.

codec: String
The codec name.

A.1.1.3.10 rtp.source.playout

rtp.source.playout (ssrc playoutDelay)
EVENT NOTIFICATION

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.

Parameters:

ssrc: string
The ssrc of the source this event notification refers to.

playoutDelay: Integer
playout delay, in milliseconds.

Full Copyright Statement

Acknowledgement

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