Network Working Group
Internet-Draft
Expires: January 12, 2001

Ott
TZI, Universitaet Bremen
Perkins
USC Information Sciences Institute
Kutscher
TZI, Universitaet Bremen
July 14, 2000

A Message Bus for Local Coordination
draft-ietf-mmusic-mbus-transport-02.txt

Status of this Memo

Copyright Notice

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 Message Bus (Mbus) provides a means to achieve the

necessary amount of coordination between co-located conferencing applications for virtually any type of conference as postulated in a a companion requirement document[11]. The Message Bus comprises two logically distinct parts: a message transport infrastructure and a set of common as well as protocol/ media/tool-specific messages along with a conference-specific addressing scheme. This document deals with message addressing, transport, and security issues and defines the message syntax for the Mbus. It does not define application oriented semantics and procedures for using the message bus. Application specific command sets and procedures for applications using the Mbus are expected to be defined in follow-up documents.

This document is intended for discussion in 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.

Table of Contents

1. Introduction

1.1 Background

The requirement specification as defined in the requirements draft[11] 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 document.

1.2 Purpose

Two components constitute the Message Bus: the (lower level) message passing mechanisms and the (higher level) messages and their semantics along with their addressing scheme.

The purpose of this document is to define the characteristics of the lower level Mbus message passing mechanism which is common to all Mbus implementations. This includes the specification of

o the generic Mbus message format;

o the addressing concept for application entities (note that addressing details are defined by the application environment);

o the transport mechanisms to be employed for conveying messages between (co-located) application entities;

o the security concept to prevent misuse of the Message Bus (as taking control of another user's conferencing environment);

o the details of the Mbus message syntax; and

o a set of mandatory application independent commands that are used for bootstrapping Mbus sessions.

1.3 Terminology for requirement specifications

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in RFC 2119[1] and indicate requirement levels for compliant Mbus implementations.

2. General Outline

The Mbus is supposed to operate in a variety of scenarios as outlined in the companion requirement document[11]. From these scenarios, the following (minimum) requirements are derived that have to be met by the Mbus design to provide a suitable local communication infrastructure.

Local coordination involves a widely varying number of entities: some messages (such as membership information, floor control notifications, dissemination conference state changes, etc.) may need to be destined for all local application entities. Messages may also be targeted at a certain application class (e.g. all whiteboards or all audio tools) or agent type (e.g. all user interfaces rather than all media engines). Or there may be any (application- or message- specific) subgrouping defining the intended recipients, e.g. messages related to media synchronization. Finally, there will be messages that are directed to a single entity, for example, specific configuration settings that a conference controller sends to a application entity or query-response exchanges between any local server and its clients.

The Mbus concept as presented here satisfies these different communication models by defining different message transport mechanisms (defined in Section 6) and by providing a flexible addressing scheme (defined in Section 4).

Furthermore, Mbus messages exchanged between application entities may have different reliability requirements (which are typically derived from their semantics). Some messages will have a rather informational character conveying ephemeral state information (which is refreshed/updated periodically), such as the volume meter level of an audio receiver entity to be displayed by its user interface agent. Certain Mbus messages (such as queries for parameters or queries to local servers) may require a response from the peer(s) thereby providing an explicit acknowledgment at the semantic level on top of the Mbus. Other messages will modify the application or conference state and hence it is crucial that they do not get lost. The latter type of message has to be delivered reliably to the recipient, whereas message of the first type do not require reliability mechanisms at the Mbus transport layer. For messages confirmed at the application layer it is up to the discretion of the application whether or not to use a reliable transport underneath.

In some cases, application entities will want to tailor the degree of reliability to their needs, others will want to rely on the underlying transport to ensure delivery of the messages -- and this may be different for each Mbus message. The Mbus message passing mechanism described in this paper provides a maximum of flexibility by providing reliable transmission achieved through transport-layer acknowledgments (in case of point-to-point communications only) as well as unreliable message passing (for unicast, local multicast, and local broadcast). We address this topic in Section 4.

Finally, accidental or malicious disturbance of Mbus communications through messages originated by applications from other users needs to be prevented. Accidental reception of Mbus messages from other users may occur if either two users share the same workstation for conferencing or are using end systems spread across the same physical network: in either case, the Mbus multicast address and the port number may match leading to reception of the other party's Mbus messages in addition to a user's own ones. Malicious disturbance may happen because of applications multicasting (e.g. at a global scope) or unicasting Mbus messages (which could contain a "conf.terminated" command). To eliminate the possibility of receiving bogus Mbus messages, the Mbus protocol contains message digests for authentication. Furthermore, the Mbus allows for encryption to ensure privacy and thus enable using the Mbus for local key distribution and other functions potentially sensitive to eavesdropping. This document defines the framework for configuring Mbus applications with regard to security parameters in Section 12.

3. Message Format

A Mbus message comprises a header and a body. The header is used to indicate how and where a message should be delivered, the body provides information and commands to the destination entity. The following information is included in the header:

The MsgDigest is a Base64-encoded (see RFC1521[5]) calculated hash value of the entire message (starting from the ProtocolID field) as described in Section 11 and Section 12.

A fixed ProtocolID field identifies the version of the message bus protocol used. The protocol defined in this document is "mbus/1.0" (case-sensitive).

A sequence number (SeqNum) is contained in each message. The first message sent by a source SHOULD have SeqNum equal to zero, and it MUST increment by one for each message sent by that source. A single sequence number is used for all messages from a source, irrespective of the intended recipients and the reliability mode selected. SeqNums are decimal numbers in ASCII representation.

The TimeStamp field is also contained in each message and SHOULD contain a decimal number representing the time at message construction in seconds since 00:00:00, UTC, January 1, 1970.

A MessageType field indicates the kind of message being sent. The value "R" indicates that the message is to be transmitted reliably and MUST be acknowledged by the recipient, "U" indicates an unreliable message which MUST NOT be acknowledged.

The SrcAddr field identifies the sender of a message. This MUST be a complete address, with all address elements specified. The addressing scheme is described in Section 4.

The DestAddr field identifies the intended recipient(s) of the message. This field MAY contain wildcards by omitting address element and hence address any number (including zero) of application entities. The addressing scheme is described in Section 4.

The AckList field comprises a list of SeqNums for which this message is an acknowledgment. See Section 5 for details.

The header is followed by the message body which contains one or more commands to be delivered to the destination entity. The syntax for a complete message is given in Message syntax (Section 7).

If multiple commands are contained within the same Mbus message payload, they MUST to be delivered to the Mbus application in the same sequence in which they appear in the message payload.

4. Addressing

Each entity on the message bus SHOULD respond to messages sent to one (or more) addresses. Addresses are sequences of address elements that are tag/value pairs. The tag and the value are separated by a colon and tag/value pairs are separated by whitespace, like this:

```
(tag:value tag:value ...)
```

The formal ABNF syntax definition for Mbus addresses and their elements is as follows:

```
mbus_address    = "(" *address_element ")"
address_element = *WSP address_tag ":" address_value *WSP
address_tag     = 1*32(ALPHA)
address_value   = 1*64(%x21-7F)
                  ; any 7-bit US-ASCII character
                  ; excluding white space
                  ; and control characters
```

Each entity has a fixed sequence of address elements constituting its address and MUST only process messages sent to addresses that either match all elements or consist of a subset of its own address elements. Each element value in this subset must match the correspoding value of the receiver's address element value. The order of address elements in an address sequence is not relevant. For example, an entity with an address of:

```
(conf:test media:audio module:engine app:rat id:4711-1@134.102.218.45)
```

will process messages sent to

```
(media:audio module:engine)
```

and

```
(module:engine)
```

but must ignore messages sent to

```
(conf:test media:audio module:engine app:rat id:123-4@134.102.218.45 foo:bar)
```

and

```
(foo:bar)
```

A message that should be processed by all entities requires an empty set of address elements.

4.1 Mandatory Address Elements

Each Mbus entity MUST provide one mandatory address element that allows to identify the entity. The element name is "id" and the value MUST be be composed of the following components:

- o The IP address of the interface that is used for sending messages to the Mbus. For IPv4 this the address in decimal dotted notation. For IPv6 the interface-ID-part of an address in textual representation as specified in [3] MUST be used. In this specification, this part is called the "host-ID".

- o An identifier ("entity-ID") that is unique within the scope of single host-ID. The entity comprises two parts. For systems where the concept of a process ID is applicable it is RECOMMENDED this identifier be composed using a process-ID and a per-process disambiguator for different Mbus entities of a process. If a process ID is not available, this part of the entity-ID may be randomly chosen (it is recommended that at least a 32 bit random number is chosen). Both numbers are represented in decimal textual form and MUST be separated by a '-' character.

Note that the entity-ID cannot be the port number of the endpoint used for sending messages to the Mbus because implementations MAY use the common Mbus port number for sending to and receiving from the multicast group (as specified in Section 6). The total identifier has the following structure:

```
id-element   = "id:" id-value
id-value     = entity-id "@" host-id
entity-id    = 1*10DIGIT "-" 1*5DIGIT
host-id      = (IPv4address / IPv6address)
```

Please refer to [3] for productions of IPv4address and IPv6address.

An example for an id element:

```
id:4711-99@134.102.218.45
```

A set of the address elements that are to be used by conferencing applications is specified in "Mbus Addresses for Conferencing" (Appendix A).

5. Reliability

While most messages are expected to be sent using unreliable transport, it may be necessary to deliver some messages reliably. Reliability can be selected on a per message basis by means of the MessageType field. Reliable delivery is supported for messages with a single recipient only; i.e., all components of the DestAddr field have to be specified. An entity can thus only send reliable messages to known addresses, i.e. it can only send reliable messages to entities that have announced their existence on the Mbus (e.g. by means of mbus.hello() messages (Section 9.1)). A sending entity MUST NOT send a message reliably if the target address is not unique. (See Transport (Section 6) for the specification of an algorithm to determine whether an address is unique.) A receiving entity MUST only process and acknowledge a reliable message if the destination address exactly matches its own source address (the destination address MUST NOT be a subset of the source address).

Disallowing reliable message delivery for messages sent to multiple destinations is motivated by simplicity of the implementation as well as the protocol. Although ACK implosions are not really an issue and losses are rare, achieving reliability for such messages would require full knowledge of the membership for each subgroup which is deemed too much effort.

Each message is tagged with a message sequence number. If the MessageType is "R", the sender expects an acknowledgment from the recipient within a short period of time. If the acknowledgment is not received within this interval, the sender SHOULD retransmit the message (with the same message sequence number), increase the timeout, and restart the timer. Messages MUST be retransmitted a small number of times (see below) before the recipient is considered to have failed. If the message is not delivered successfully, the sending application is notified. In this case, it is up to this application to determine the specific action(s) (if any) to be taken.

Reliable messages are acknowledged by adding their SeqNum to the AckList field of a message sent to the originator of the reliable message. Multiple acknowledgments MAY be sent in a single message. It is possible to either piggy-back the AckList onto another message sent to the same destination, or to send a dedicated acknowledgment message, with no commands in the message payload part.

The precise procedures are as follows:

Sender: A sender A of a reliable message M to receiver B SHOULD transmit the message via multicast or via unicast, keep a copy of M, initialize a retransmission counter N to '1', and start a retransmission timer T (initialized to T_r). If an acknowledgment is received from B, timer T MUST BE cancelled and the copy of M is discarded. If T expires, the message M SHOULD BE retransmitted, the counter N SHOULD BE incremented by one, and the timer SHOULD BE restarted (set to N*T_r). If N exceeds the retransmission threshold N_r, the transmission is assumed to have failed, further retransmission attempts MUST NOT be undertaken, the copy of M SHOULD BE discarded, and the sending application SHOULD BE notified.

Receiver: A receiver B of a reliable message from a sender A SHOULD acknowledge receipt of the message within a time period T_c < T_r. This MAY be done by means of a dedicated acknowledgment message or by piggy-backing the acknowledgment on another message addressed only to A.

Receiver optimization: In a simple implementation, B may choose to immediately send a dedicated acknowledgment message. However, for efficiency, it could add the SeqNum of the received message to a sender-specific list of acknowledgments; if the added SeqNum is the first acknowledgment in the list, B SHOULD start an acknowledgment timer TA (initialized to T_c). When the timer expires, B SHOULD create a dedicated acknowledgment message and send it to A. If B is to transmit another Mbus message addressed only to A, it should piggy-back the acknowledgments onto this message and cancel TA. In either case, B should store a copy of the acknowledgment list as a single entry in the per- sender copy list, keep this entry for a period T_k, and empty the acknowledgment list. In case any of the messages kept in an entry of the copy list is received again from A, the entire acknowledgment list stored in this entry is scheduled for (re-)transmission following the above rules.

Constants and Algorithms: The following constants and algorithms SHOULD be used by implementations:

T_r=100ms

N_r=3

T_c=70ms

T_k=((N_r)*(N_r+1)/2)*T_r

6. Transport

All messages are transmitted as UDP messages with two ways of sending messages being possible:

1. Local multicast (host-local or link-local, see Mbus configuration (Section 12)) to a fixed, yet to be assigned (see Section 14) link-local address of the administratively scoped multicast space as described in RFC 2365[10]. There will also be a fixed, registered port number that all Mbus entities MUST use. Until the address and port numer are assigned, 224.255.222.239 is used as the multicast address and 47000 (decimal) as the port number.

2. Directed unicast (via UDP) to the port of a specific application. This still requires the DestAddr field to be filled in properly. Directed unicast is intended for situations where node local multicast is not available. It MAY also be used by Mbus implementations for delivering messages addressed at a single application entity only -- the address of which the Mbus implementation has learned from other message exchanges before. Every Mbus entity SHOULD use a unique endpoint address for every message it sends to the Mbus multicast group or to individual receiving entities. A unique endpoint address is a tuple consisting of the entity's IP address and a port number, where the port number is different from the standard Mbus port number (yet to be assigned, see Section 14). When multicast is available, messages MUST only be sent via unicast if the Mbus target address is unique and if the sending entity can verify that the receiving entity uses a unique endpoint address. The latter can be verified by considering the last message received from that entity. (Note that several Mbus entities, say within the same process, may share a common transport address; in this case, the contents of the destination address field is used to further dispatch the message. Given the definition of "unique endpoint address" above the use of a shared endpoint address and a dispatcher still allows other Mbus entities to send unicast messages to one of the entities that share the endpoint address. So this can be considered an implementation detail.) When multicast is not available messages can be sent via unicast but all messages that do not contain a unique target address MUST be sent to all known entities via unicast. Messages with an empty target address list MUST always be sent to all Mbus entities (via multicast if available).
   The following algorithm can be used by sending entities to determine whether a Mbus address is unique considering the current set of Mbus entities:

```
let ta=the target address;
iterate through the set of all
currently known Mbus addresses {
  let ti=the address in each iteration;
  count the addresses for which
  the predicate isSubsetOf(ta,ti) yields true;
}
```

If the count of matching addresses is exactly 1 the address is unique. The following algorithm can be used for the predicate isSubsetOf, that checks whether the second message matches the first according to the rules specified in Section 4. (A match means that a receiving entity that uses the second Mbus address must also process received messages with the first address as a target address.

```
isSubsetOf(addr a1,a2) yields true, iff
  every address element of a1 is contained
  in a2's address element list
```

An address element is contained in an address element list if the list contains an element that provides same values for the two address element fields key and value.

If a single application system is distributed across several co-located hosts, link local scope SHOULD be used for multicasting Mbus messages that potentially have recipients on the other hosts. The Mbus protocol is not intended (and hence deliberately not designed) for communication between hosts not on the same link.

Since messages are transmitted in UDP datagrams, a maximum size of 64 KBytes MUST NOT be exceeded. It is RECOMMENDED that applications using a non host-local scope do not exceed a message size of the network's MTU.

7. Message Syntax

7.1 Message Encoding

All messages MUST use the UTF-8 character encoding. Note that US ASCII is a subset of UTF-8 and requires no additional encoding, and that a message encoded with UTF-8 will not contain zero bytes.

Each Message MAY be encrypted using a secret key algorithm as defined in Section 11.

7.2 Message Header

A message starts with the header. The first field in the header is the message digest calculated using a keyed hash algorithm as described in Section 11 followed by a newline character. The other fields in the header are separated by white space characters, and followed by a newline. The format of the header is as follows:

```
msg_header = MsgDigest LF "mbus/1.0" 1*WSP SeqNum 1*WSP TimeStamp 1*WSP
             MessageType 1*WSP SrcAddr 1*WSP DestAddr 1*WSP AckList
```

The header fields are explained in Message Format (Section 3). Here are the ABNF syntax definitions for the header fields:

```
MsgDigest   = base64
SeqNum      = 1*DIGIT
TimeStamp   = 1*DIGIT
MessageType = "R" / "U"
ScrAddr     = mbus_address
DestAddr    = mbus_address
AckList     = "(" *(1*DIGIT)) ")"
```

The syntax definition of a complete message is as follows:

```
mbus_message = msg_header *1(LF msg_payload)
msg_payload  = mbus_command *(LF mbus_command)
```

See Figure 18 for the definition a Mbus command.

7.3 Command Syntax

The header is followed by zero, or more, commands to be delivered to the application(s) indicated by the DestAddr field. Each message comprises a command followed by a list of zero, or more, parameters, and is followed by a newline.

```
command ( parameter parameter ... )
```

Syntactically, the command name MUST be a ‘symbol’ as defined in the following table. The parameters MAY be any data type drawn from the following table:

DataType	Syntax	Description
val	(Integer / Float / String / List Symbol Data)	a value can be of one of these types
Integer	"-" 1*DIGIT	
Float	"-" 1*DIGIT "." 1*DIGIT	
String	DQUOTE *CHAR DQUOTE	See below for escape characters
List	"(" *(val *(WSP val)) ")"	
Symbol	ALPHA *(ALPHA / DIGIT / "_" / "-" / ".")	A predefined protocol value
Data	"<" *base64 ">"	Opaque Data

Boolean values are encoded as an integer, with the value of zero representing false, and non-zero representing true (as in the ‘C’ programming language).

String parameters in the payload MUST be enclosed in the double quote (’’) character. Within strings, the escape character is the backslash (\), and the following escape sequences are defined:

Escape Sequence	Meaning
\\	\
\"	"
\n	newline

List parameters do not have to be homogeneous lists, i.e. they can contain parameters of varying types.

Opaque data is represented as Base64-encoded (see RFC1521[5]) character strings surrounded by "< " and "> "

The ABNF syntax definition for Mbus commands is as follows:

```
mbus_command = command_name arglist
command_name = ALPHA *(ALPHA / DIGIT / "_" / ".")
arglist      = "(" *(*WSP parameter *WSP) ")"
parameter    = Integer / Float / String / List
               Symbol / Data
```

Command names SHOULD 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 Addressing (Section 4) 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 SHOULD be unambiguous in a way that it is possible to interpret or ignore 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 Messages (Section 9).

8. Awareness of other Entities

Before Mbus entities can communicate with one another, 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. Furthermore entities need to be able to to notice the failure (or leaving) of other entities.

Any Mbus entity MUST 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: Entities should always become aware of other entities independent of the order of starting.

Each Mbus entity MUST maintain the number of Mbus session members and continously update this number according to any observed changes. The mechanisms of how the existence and the leaving of other entities can be detected are dedicated Mbus messages for entity awareness: mbus.hello (Section 9.1) and mbus.bye (Section 9.2). Each Mbus protocol implementation MUST periodically send mbus.hello messages that are used by other entities to monitor the existence of that entity. If an entity has not received mbus.hello messages for a certain time (see Section 8.2) from an entity the respective entity is considered to have left the Mbus and MUST be excluded from the set of currently known entities. Upon the reception of a mbus.bye messages the respective entity is considered to have left the Mbus as well and MUST be excluded from the set of currently known entities immediately.

Each Mbus entity MUST send hello messages after startup to the Mbus. After transmission of the hello message, it shall start a timer after the expiration of which the next hello message is to be transmitted. Transmission of hello messages MUST NOT be stopped unless the entity detaches from the Mbus. The interval for sending hello messages is depending on the current number of entities in a Mbus group and can thus change dynamically in order to avoid congestion due to many entities sending hello messages at a constant high rate.

Section 8.1 specifies the calculation of hello message intervals that MUST be used by protocol implementations. Using the values that are calculated for obtaining the current hello message timer, the timeout for received hello messages is calculated in Section 8.2. Section 9 specifies the command synopsis for the corresponding Mbus messages.

8.1 Hello Message Transmission Interval

Since Mbus sessions may vary in size concerning the number of entities care must be taken to allow the Mbus protocol to scale well over different numbers of entities automatically. The average rate at which hello messages are received would increase linearly to the number of entities in a session if the sending interval was set to a fixed value. Given a interval of 1 second this would mean that an entity taking part in an Mbus session with n entities would receive n hello messages per second. Assuming all entities resided on one host this would lead to n*n messages that have to be processed per second -- which is obviously not a viable solution for larger groups. It is therefore necessary to deploy dynamically adapted hello message intervals taking varying numbers of entities into account. In the following we specify an algorithm that MUST be used by implementations to calculate the interval for hello messages considering the observed number of Mbus entities.

The algorithm features the following characteristics:

- o The number of hello messages that are received by a single entity in a certain time unit remains approximately constant as the number of entities changes.

- o The effective interval that is used by a specific Mbus entity is randomized in order to avoid unintentional synchronization of hello messages within a Mbus session. The first hello message of an entity is also delayed by a certain random amount of time.

- o A timer reconsideration mechanism is deployed in order to adapt the interval more appropriately in situations where a rapid change of the number of entities is observed. This is useful when an entity joins an Mbus sessions and is still learning of the existence of other entities or when a larger number of entities leaves the Mbus at once.

8.1.1 Calculating the Interval for Hello Messages

The following names for values are used in the calculation specified below (all time values in milliseconds):

hello_p: The last time a hello message has been sent by a Mbus entity.

hello_now: The current time

hello_d: The deterministic calculated interval between hello messages.

hello_e: The effective (randomized) interval between hello messages.

hello_n: The time for the next scheduled transmission of a hello message.

entities_p: The numbers of entities at the time hello_n has been last recomputed.

entities: The number of currently known entities.

The interval between hello messages MUST be calculated as follows:

The number of currently known entities is multiplied by c_hello_factor, yielding the interval between hello messages in milliseconds. This is the deterministic calculated interval, denominated hello_d. The minimum value for hello_d is c_hello_min. Thus hello_d=max(c_hello_min,c_hello_factor * entities). Section 8 provides a specification of how to obtain the number of currently known entities. Section 10 provides values for the constants c_hello_factor and c_hello_min.

The effective interval hello_e that is to be used by individual entities is calculated by multiplying hello_d with a randomly chosen number between c_hello_dither_min and c_hello_dither_max (see Section 10).

hello_n, the time for the next hello message in milliseconds is set to hello_e + hello_now.

8.1.2 Initialization of Values

Upon joining a session a protocol implementation sets hello_p, hello_now to 0 and entities, entities_p to 1 (the current Mbus entity itself) and then calculates the time for the next hello message as specified in Section 8.1.1. The next hello message is scheduled for transmission at hello_n.

8.1.3 Adjusting the Hello Message Interval when the Number of Entities increases

When the existence of a new entity is observed by a protocol implementation the number of currently known entities is updated. No further action concerning the calculation of the hello message interval is required. The reconsideration of the timer interval takes place when the current timer for the next hello message expires (see Section 8.1.5).

8.1.4 Adjusting the Hello Message Interval when the Number of Entities decreases

Upon realizing that an entity has left the Mbus the number of currently known entities is updated and the following algorithm should be used to reconsider the timer interval for hello messages:

1. The value for hello_n is updated by setting hello_n to hello_now + (entities/entities_p)*(hello_n - hello_now)

2. The value for hello_p is updated by setting hello_p to hello_now - (entities/entities_p)*(hello_now - hello_p)

3. The currently active timer for the next hello messages is cancelled and a new timer is started for hello_n.

4. entities_p is set to entities.

8.1.5 Expiration of hello timers

When the hello message timer expires, the protocol implementation MUST perform the following operations:

The hello interval hello_e is computed as specified in Section 8.1.1.

If

1. hello_e + hello_p is less than or equal to hello_now, a hello message is transmitted. hello_p is set to hello_now, hello_e is calculated again as specified in Section 8.1.1 and hello_n is set to hello_e + hello_now.

2. else if hello_e + hello_p is greater than hello_now, hello_n is set to hello_e + hello_p. A new timer for the next hello message is started to expire at hello_n. No hello message is transmitted.

entities_p is set to entities.

8.2 Calculating the Timeout for Hello Messages

Whenever an Mbus entity has not heard for a time span of c_hello_dead*(hello_d*c_hello_dither_max) milliseconds from another Mbus entity it may consider this entity to have failed (or have quit silently). The number of the currently known entities MUST be updated accordingly. Note that no need for any further action is necessarily implied from this observation.

Section 8.1.1 specifies how to obtain hello_d. Section 10 defines values for the constants c_hello_dead and c_hello_dither_max.

9. Messages

This section defines some basic application independent messages that MUST be understood by all implementations. This specification does not contain application specific messages which are to be defined outside of the basic Mbus protocol specification.

An 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 commend set to implement the aforementioned concepts is presented in the following sections.

9.1 mbus.hello

```
Syntax:
mbus.hello(parameters...)

Parameters: see below
```

HELLO messages MUST be 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 and can thus calculate the current number of entities.

HELLO messages MUST be sent periodically in dynamically calculated intervals as specified in Section 8.

Upon startup the first HELLO message MUST be sent after a delay hello_delay, where hello_delay be a randomly chosen number between 0 and c_hello_min (see Section 10).

9.2 mbus.bye

```
Syntax:

Parameters: - none -
```

An Mbus entity that is about to terminate (or "detach" from the Mbus) SHOULD announce this by transmitting a BYE message.

The BYE message MUST be sent unreliably to all receivers.

9.3 mbus.ping

Syntax:

Parameters: - none -

mbus.ping can be used to solicit other entities to signal their existence by replying with a mbus.hello message. Each protocol implementation MUST understand mbus.ping and reply with a mbus.hello message. The reply hello message MUST be delayed for hello_delay milliseconds, where hello_delay be a randomly chosen number between 0 and c_hello_min (see Section 10).

As specified in Section 9.1 hello messages MUST be sent unreliably to all Mbus entities. An entity that replies to mbus.ping with mbus.hello should stop any outstanding timers for hello messages after sending the hello message and schedule a new timer event for the subsequent hello message. (Note that using the variables and the algorithms of Section 8.1.1 this can be achieved by setting hello_p to hello_now.)

mbus.ping allows a new entity to quickly check for other entities without having to wait for the regular individual hello messages. By specifying a target address the new entity can restrict the solicitation for hello messages to a subset of entities it is interested in.

9.4 mbus.quit

Syntax:
mbus.quit()

Parameters: - none -

The QUIT message is used to request other entities to terminate themselves (and detach from the Mbus). Whether this request is honoured by receiving entities or not is up to the discretion of the application.

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

9.5 mbus.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.

9.6 mbus.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.

10. Constants

The following values for timers and counters mentioned in this document SHOULD be used by implementations:

Timer / Counter	Value	Unit
c_hello_factor	200	-
c_hello_min	1000	milliseconds
c_hello_dither_min	0.9	-
c_hello_dither_max	1.1	-
c_hello_dead	5	-

11. Mbus Security

11.1 Security Model

In order to prevent accidental or malicious disturbance of Mbus communications through messages originated by applications from other users message authentication is deployed (Section 11.2). For each message a digest is calculated based on the value of a shared secret key value. Receivers of messages can check if the sender belongs to the same Mbus security domain by re-calculating the digest and comparing it to the received value. Only if both values are equal the messages must be processed further. In order to allow different simultaneous Mbus sessions at a given scope and to compensate defective implementations of host local multicast ([18]) message authentication MUST be provided by conforming implementations.

Privacy of Mbus message transport can be achieved by optionally using symmetric encryption methods (Section 11.3). Each message can be encrypted using an additional shared secret key and a symmetric encryption algorithm. Encryption is OPTIONAL for applications, i.e. it is allowed to configure an Mbus domain not to use encryption. But conforming implementations MUST provide the possibility to use message encryption (see below).

Message authentication and encryption can be parameterized by certain values, e.g. by the algorithms to apply or by the keys to use. These parameters (amongst others) are defined in an Mbus configuration entity that is accessible to all Mbus entities that participate in an Mbus session. In order to achieve interoperability conforming implementations SHOULD consider the given Mbus configuration entity. Section 12 defines the mandatory and optional parameters as well as storage procedures for different platforms. Only in cases where none of the options for configuration entities mentioned in Section 12 is applicable alternative methods of configuring Mbus protocol entities MAY be deployed.

11.2 Message Authentication

Either MD5 [14] or SHA-1 [15] SHOULD be used for message authentication codes (MACs). An implementation MAY provide SHA-1, whereas MD5 MUST be implemented. To generate keyed hash values the algorithm described in RFC2104[4] MUST be applied with hash values truncated to 96 bits (12 bytes). The resulting hash values MUST be Base64 encoded (16 characters). The HMAC algorithm works with both, MD5 and SHA-1.

HMAC values, regardless of the algorithm, MUST therefore always consist of 16 Base64-encoded characters.

Hash keys MUST have a length of 96 bit (12 bytes), that are 16 Base64-encoded characters.

11.3 Encryption

Either DES, 3DES (triple DES) or IDEA SHOULD be used for encryption. Encryption MAY be neglected for applications, e.g. in situations where license regulations, export or encryption laws would be offended otherwise. However, the implementation of DES is RECOMMENDED as a baseline. DES implementations MUST use the DES Cipher Block Chaining (CBC) mode. For algorithms requiring en/decryption data to be padded to certain boundaries octets with a value of 0 SHOULD be used for padding characters. The padding characters MUST be appended after calculating the message digest when encoding and MUST be erased before recalculating the message digest when decoding. IDEA uses 128-bit keys (24 Base64-encoded characters). DES keys (56 bits) MUST be encoded as 8 octets as described in RFC1423[12], resulting in 12 Base64-encoded characters.

The mandatory subset of algorithms that MUST be provided by implementations is DES and MD5.

See Section 12 for a specification of notations for Base64-strings.

12. Mbus Configuration

An implementation MUST be configurable by the following parameters:

Configuration version

The version number of the given configuration entity. Version numbers allow implementations to check if they can process the entries of a given configuration entity. Version number are integer values. The version number for the version specified here is 1.

Encryption key

The secret key used for message encryption.

Hash key

The hash key used for message authentication.

Scope

The Internet scope to be used for sent messages.

The upper parameters are mandatory and MUST be present in every Mbus configuration entity.

The following parameters are optional. When they are present they MUST be honoured but when they are not present implementations SHOULD fall back to the predefined default values (as defined in Transport (Section 6)):

Address

The non-standard multicast address to use for message transport.

Port

The non-standard port number to use for message transport.

Two distinct facilities for parameter storage are considered: For Unix-like systems a configuration file SHOULD be used and for Windows-95/98/NT/2000 systems a set of registry entries is defined that SHOULD be used.

The syntax of the values for the respective parameter entries remains the same for both configuration facilities. The following defines a set of ABNF (see RFC2234[13]) productions that are later referenced for the definitions for the configuration file syntax and registry entries:

```
algo-id                 =    "NOENCR" / "DES" / "3DES" / "IDEA" /
                             "HMAC-MD5-96" / "HMAC-SHA1-96"
scope                   =    "HOSTLOCAL" / "LINKLOCAL"
key                     =    base64string
version_number          =    1*10DIGIT
base64string            =    *(ALPHA / DIGIT / "+" / "/" / "=")
key_value               =    "(" algo-id "," key ")"
ipv4_addr               =    ipv4_octet 3*3("." ipv4_octet)
ipv4_octet              =    1*3DIGIT
port                    =    1*5DIGIT
```

A key entry MUST be specified using this notation:

```
"("algo-id","base64string")"
```

algo-id is one of the character strings specified above. For algo-id=‘‘NOENCR’’ the other fields are ignored. The de- limiting commas MUST always be present though.

A Base64 string consists of the characters defined in the Base64 char-set (see RFC1521[5]) including all eventual padding characters, i.e. the length of Base64-string is always a multiple of 4.

The version_number parameter specifies a version number for the used configuration entity.

12.1 File based parameter storage

The file name for a Mbus configuration file is ".mbus" in the user’s home-directory. If an environment variable called MBUS is defined implementations SHOULD interpret the value of this variable as a fully qualified file name that is to be used for the configuration file. Implementations MUST ensure that this file has appropriate file permissions that prevent other users to read or write it. The file MUST exist before a conference is initiated. Its contents MUST be UTF-8 encoded and MUST be structured as follows:

```
mbus-file     =    mbus-topic LF *(entry LF)
mbus-topic    =    "[MBUS]"
entry         =     1*(version_info / hashkey_info
                       / encryptionkey_info / scope_info
                       / port_info / address_info)
version_info  =    "CONFIG_VERSION=" version_number
hashkey_info  =    "HASHKEY=" key_value
encrkey_info  =    "ENCRYPTIONKEY=" key_value
scope_info    =    "SCOPE=" scope
port_info     =    "PORT=" port
address_info  =    "ADDRESS=" ipv4_addr
```

The following entries are defined: CONFIG_VERSION, HASHKEY, ENCRYPTIONKEY, SCOPE, PORT, ADDRESS.

The entries CONFIG_VERSION, HASHKEY and ENCRYPTIONKEY are mandatory, they MUST be present in every Mbus configuration file. The order of entries is not significant.

An example Mbus configuration file:

```
[MBUS]
CONFIG_VERSION=1
HASHKEY=(HMAC-MD5-96,MTIzMTU2MTg5MTEy)
ENCRYPTIONKEY=(DES,MTIzMTU2MQ==)
SCOPE=HOSTLOCAL
ADDRESS=224.255.222.239
PORT=47000
```

12.2 Registry based parameter storage

For systems lacking the concept of a user’s home-directory as a place for configuration files the suggested database for configuration settings (e.g. the Windows9x-, Windows NT-, Windows 2000-registry) SHOULD be used. The hierarchy for Mbus related registry entries is as follows:

```
HKEY_CURRENT_USER\Software\Mbone Applications\Mbus
```

The entries in this hierarchy section are:

Name	Type	ABNF production
CONFIG_VERSION	DWORD	version_number
HASHKEY	String	key_value
ENCRYPTIONKEY	String	key_value
SCOPE	String	scope
ADDRESS	String	ipv4_addr
PORT	DWORD	port

The same syntax for key values as for the file based configuration facility MUST be used.

13. Security Considerations

The Mbus security mechanismns are specified in Section 11.1.

It should be noted that the Mbus transport specification defines a mandatory baseline set of algorithms that have to be supported by implementations. This baseline set does not neccessarily provide the best security due to the cryptographic weaknesses of the individual algorithms. For example, it has been stated in [4] that MD5 had been shown to be vulnerable to collision search attacks (although this was believed not to compromise the use of MD5 within HMAC generation). However, SHA-1 is usually considered to be the cryptographically stronger function ([16]).

Similar remarks can be made on the encryption functions. The base specification requires DES, an algorithm that has shown to be vulnerable to brute-force attacks ([16], [17]).

We do not consider the well-known weaknesses of the mentioned algorithms a problem:

- The problem of receiving unauthenticated messages is considered to be the main security threat for Mbus communication. We believe that HMAC-MD5 is sufficiently secure as a baseline algorithm. For application requiring special security concerning authentication of messages there is the option of using implementations that implement SHA-1.

- Encryption is optional anyway, i.e. users can decide to have their implementations sending clear text Mbus messages. Given the local nature of Mbus communication this is feasible for most use cases. In case the base DES encryption is not considered sufficient there is still the possibility to use implementations that implement 3DES or IDEA.

However, application developers should be aware of incorrect IP implementations that do not conform to RFC 1122[2] and do send datagrams with TTL values of zero, resulting in Mbus messages sent to the local network link although a user might have selected host local scope in the Mbus configuration. In these cases the use of encryption SHOULD be considered if privacy is desired.

14. IANA Considerations

The IANA is requested to assign a port number and a multicast address. For the time being the tentative multicast address 224.255.222.239 and the port number 47000 (decimal) SHOULD be used.

References

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

[2] Braden, R., "Requirements for Internet Hosts -- Communication Layers", RFC 1122, October 1989.

[3] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 2373, July 1998.

[4] Krawczyk, H., Bellare, M. and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[5] Borenstein, N. and N. Freed, "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies", RFC 1521, September 1993.

[6] Handley, M., Crowcroft, J., Bormann, C. and J. Ott, "The Internet Multimedia Conferencing Architecture", Internet Draft draft-ietf-mmusic-confarch-02.txt, October 1999.

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

[8] Handley, M., Schulzrinne, H., Schooler, E. and J. Rosenberg, "SIP: Session Initiation Protocol", RFC 2543, March 1999.

[9] Handley, M. and V. Jacobsen, "SDP: Session Description Protocol", RFC 2327, April 1998.

[10] Meyer, D., "Administratively Scoped IP Multicast", RFC 2365, July 1998.

[11] Ott, J., Perkins, C. and D. Kutscher, "Requirements for Local Conference Control", Internet Draft draft-ietf-mmusic-mbus-req-00.txt, December 1999.

[12] Balenson, D., "Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers", RFC 1423, February 1993.

[13] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.

[14] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

[15] U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology, "Secure Hash Standard", FIPS PUB 180-1, April 1995.

[16] Schneier, B., "Applied Cryptography", Edition 2, Publisher John Wiley & Sons, Inc., 1996.

[17] distributed.net, "Project DES", WWW http://www.distributed.net/des/, 1999.

[18] Microsoft, "BUG: Winsock Sends IP Packets with TTL 0", WWW http://support.microsoft.com/support/kb/articles/Q138/2/68.asp, March 1999 .

Authors' Addresses

Joerg Ott
TZI, Universitaet Bremen
Bibliothekstr. 1
Bremen 28359
Germany

Phone: +49.421.201-7028
Fax: +49.421.218-7000
EMail: jo@tzi.uni-bremen.de

Colin Perkins
USC Information Sciences Institute
4350 N. Fairfax Drive #620
Arlington VA 22203
USA

EMail: csp@isi.edu

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

Phone: +49.421.218-7595
Fax: +49.421.218-7000
EMail: dku@tzi.uni-bremen.de

Appendix A. Mbus Addresses for Conferencing

For conferencing application 5 address element keys are predefined (in addition to the mandatory element key "id"):

conf	conference identifier
media	media type processed by application
module	module type of Mbus entity in a application
app	application name

The conf element is used to designate the name of a conference in order to distinguish between entities that are present in more than one conference. See Transport (Section 6) for further notes concerning multiple presences using the Mbus.

The media element identifies the type of media processed by an application. Currently defined values are:

audio	An RTP audio stream
video	An RTP video stream
workspace	A shared workspace
whiteboard	A shared whiteboard
editor	A shared text editor
sap	A session announcement tool, using SAP
sip	A session invitation tool, using SIP
h323	An ITU-T H.323 conference controller
rtsp	An RTSP session controller
control	A local coordination entity

Other values are likely to be defined at a later date.

The module element defines a logical part of an application. The value ‘ui’ denotes the user-interface of an application, and the value ‘engine’ defines a media/protocol engine, and ‘transcoder’ defines a media transcoder. Other values may be defined in future.

The app element identifies the application being used (e.g.: rat, vic, etc.).

The instance element is used to distinguish several instances of the same application. This is a per-instance-unique identifier, which is not necessarily an integer. Many Unix applications will use the process-id (PID) number, although this is not a requirement. Note that if an end system is spread across several hosts, the instance MUST NOT be the process-id, unless e.g.. the host name or its IP address are included as well. Section 9 defines a bootstrap procedure ensuring that entities can track the abandoning and restarting of application instances as long as unique instance values are being used.

The following examples illustrate how to make use of the addresses:

(conf:test media:audio module:ui app:rat id:4711-99@134.102.218.45)	The user interface of the rat application with the given id is taking part in conference test
(media:workspace module:ui)	The user interfaces of all workspace applications
(media:audio)	All audio applications
(app:rat)	All instances of the rat application
()	All entities

Acknowledgement

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