                  A Message Bus for Local Coordination

Status of this Memo

Copyright Notice

Abstract

   The local Message Bus (Mbus) is a light-weight message-oriented
   coordination protocol for group communication between application
   components. The Mbus provides automatic location of communication
   peers, subject based addressing, reliable message transfer and
   different types of communication schemes. The protocol is layered on
   top of IP multicast and is specified for IPv4 and IPv6. The IP
   multicast scope is limited to link-local multicast. This document
   specifies the Mbus protocol, i.e., message syntax, addressing and
   transport mechanisms.

Table of Contents

1. Introduction

   The implementation of multiparty multimedia conferencing systems is
   one example where a simple coordination infrastructure can be useful:
   In a variety of conferencing scenarios, a local communication channel
   can provide 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 application entities, e.g., to 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 an end system.

   Conferencing systems such as IP telephones can also be viewed as
   components of a distributed system and can thus be integrated into a
   group of application modules: For example, an IP telephony call that
   is conducted with a stand-alone IP telephone can be dynamically
   extended to include media engines for other media types using the
   coordination function of an appropriate coordination mechanism.
   Different individual conferencing components can thus be combined to
   build a coherent multimedia conferencing system for a user.

   Other possible scenarios include the coordination of application
   components that are distributed on different hosts in a network, for
   example, so-called Internet appliances.

1.1 Mbus Overview

   Local coordination of application components requires a number of
   different interaction models: some messages (such as membership
   information, floor control notifications, dissemination of conference
   state changes, etc.) may need to be sent to 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 may be messages that are directed at a single entity:
   for example, specific configuration settings that a conference
   controller sends to a particular application entity, or query-
   response exchanges between any local server and its clients.

   The Mbus protocol as defined here satisfies these different
   communication needs 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 transient
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 messages 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 specified in this document 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 host for using Mbus
applications or if they are using Mbus applications that are spread
across the same network link: in either case, the used Mbus multicast
address and the port number may be identical leading to reception of
the other party's Mbus messages in addition to the user's own ones.
Malicious disturbance may happen because of applications multicasting
(e.g., at a global scope) or unicasting Mbus messages. To eliminate
the possibility of processing unwanted 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.

1.2 Purpose of this Document

   Three components constitute the message bus: the low level message
   passing mechanisms, a command syntax and naming hierarchy, and the
   addressing scheme.

   The purpose of this document is to define the protocol mechanisms 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
      concrete addressing schemes are to be defined by application-
      specific profiles);

   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 (such 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 Areas of Application

   The Mbus protocol can be deployed in many different application
   areas, including but not limited to:

   Local conference control: In the Mbone community a model has arisen
      whereby a set of loosely coupled tools are used to participate in
      a conference. A typical scenario is that audio, video, and shared
      workspace functionality is provided by three separate tools
      (although some combined tools exist). This maps well onto the
      underlying RTP [8] (as well as other) media streams, which are
      also transmitted separately. Given such an architecture, it is
      useful to be able to perform some coordination of the separate
      media tools. For example, it may be desirable to communicate
      playout-point information between audio and video tools, in order
      to implement lip-synchronization, to arbitrate the use of shared
      resources (such as input devices), etc.

      A refinement of this architecture relies on the presence of a
      number of media engines which perform protocol functions as well
      as capturing and playout of media. In addition, one (or more)

(separate) user interface agents exist that interact with and
control their media engine(s). Such an approach allows
flexibility in the user-interface design and implementation, but
obviously requires some means by which the various involved agents
may communicate with one another. This is particularly desirable
to enable a coherent response to a user's conference-related
actions (such as joining or leaving a conference).

Although current practice in the Mbone community is to work with a
loosely coupled conference control model, situations arise where
this is not appropriate and a more tightly coupled wide-area
conference control protocol must be employed. In such cases, it
is highly desirable to be able to re-use the existing tools (media
engines) available for loosely coupled conferences and integrate
them with a system component implementing the tight conference
control model. One appropriate means to achieve this integration
is a communication channel that allows a dedicated conference
control entity to "remotely" control the media engines in addition
to or instead of their respective user interfaces.

Control of device groups in a network: A group of devices that are
connected to a local network, e.g., home appliances in a home
network, require a local coordination mechanism. Minimizing
manual configuration and the the possibility to deploy group
communication will be useful in this application area as well.

1.4 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. Common Formal Syntax Rules

This section contains definitions of common ABNF [13] syntax elements
that are later referenced by other definitions in this document:

```
    base64          = base64_terminal /
                      ( 1*(4base64_CHAR) [base64_terminal] )

    base64_char     = UPALPHA / LOALPHA / DIGIT / "+" / "/"
                      ;; Case-sensitive

    base64_terminal = (2base64_char "==") / (3base64_char "=")

    UPALPHA         = %x41-5A               ;; Uppercase: A-Z
```

```
      LOALPHA          =  %x61-7A              ;; Lowercase: a-z


      ALPHA            =  %x41-5A / %x61-7A    ; A-Z / a-z

      CHAR             =  %x01-7E
                            ; any 7-bit US-ASCII character,
                             excluding NUL and delete

      OCTET            =  %x00-FF
                            ; 8 bits of data

      CR               =  %x0D
                            ; carriage return

      CRLF             =  CR LF
                            ; Internet standard newline

      DIGIT            =  %x30-39
                            ; 0-9

      DQUOTE           =  %x22
                            ; " (Double Quote)

      HTAB             =  %x09
                            ; horizontal tab

      LF               =  %x0A
                            ; linefeed

      LWSP             =  *(WSP / CRLF WSP)
                            ; linear white space (past newline)

      SP               =  %x20
                            ; space

      WSP              =  SP / HTAB
                            ; white space
```

   Taken from RFC 2234 [13] and RFC 2554 [14].

3. Message Format

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

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 set SeqNum 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. The value range of a sequence number is (0,4294967295).
An implementation MUST re-set its sequence number to 0 after
reaching 4294967295. Implementations MUST take into account that
the SeqNum of other entities may wrap-around.

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 of the message
construction in milliseconds 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 be wildcarded by omitting address
elements 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 7 for details.

The header is followed by the message body which contains zero or
more commands to be delivered to the destination entity. The syntax
for a complete message is given in Section 5.

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 in the message has a unique Mbus address that is used to
   identify the entity. Mbus 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   = "(" *WSP *1address_list *WSP ")"
      address_list   = address_element
                        / address_element 1*WSP address_list

      address_element = address_tag ":" address_value

      address_tag   = 1*32(ALPHA)

      address_value   = 1*64(%x21-27 / %x2A-7E)
                          ; any 7-bit US-ASCII character
                          ; excluding white space, delete,
                          ; control characters, "(" and ")"

   Note that this and other ABNF definitions in this document use the
   non-terminal symbols defined in Section 2.

   An address_tag MUST be unique within an Mbus address, i.e., it MUST
   only occur once.

   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. The order of address elements in an address sequence is
   not relevant. Two address elements match if both their tags and
   their values are equivalent. Equivalence for address element and
   address value strings means that each octet in the one string has the
   same value as the corresponding octet in the second string. For
   example, an entity with an address of:

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

   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@192.168.1.1
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 it to identify the entity. The element tag 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 is the address in dotted decimal
    notation. For IPv6 the interface-ID-part of the node's link-local
    address in textual representation as specified in RFC 2373 [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 a
    single host-ID. The entity comprises two parts. For systems
    where the concept of a process ID is applicable it is RECOMMENDED
    that 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 '-' (ASCII x2d) 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 complete syntax definition for the entity identifier is as
follows:

```
     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 the productions of IPv4address and IPv6address.

   An example for an id element:

```
     id:4711-99@192.168.1.1
```

5. Message Syntax

5.1 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.

5.2 Message Header

   The fields in the header are separated by white space characters,
   and followed by CRLF. The format of the header is as follows:

```
   msg_header = "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:

```
     SeqNum      = 1*10DIGIT      ; numeric range 0 - 2^32-1

     TimeStamp   = 1*13DIGIT

     MessageType = "R" / "U"

     ScrAddr     = mbus_address

     DestAddr    = mbus_address
```

        AckList   = "(" *WSP *1(1*DIGIT *(1*WSP 1*10DIGIT)) *WSP ")"

        See Section 4 for a definition of "mbus_address".

    The syntax definition of a complete message is as follows:

        mbus_message = msg_header *1(CRLF msg_payload)

        msg_payload = mbus_command *(CRLF mbus_command)

    The definition of production rules for an Mbus command is given in
    Section 5.3.

5.3 Command Syntax

    The header is followed by zero, one, or more, commands to be
    delivered to the Mbus entities indicated by the DestAddr field. Each
    command consists of a command name that is followed by a list of
    zero, or more parameters and is terminated 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:

        val   = Integer / Float / String / List /
                          Symbol / Data

        Integer   = *1"-" 1*DIGIT

        Float   = *1"-" 1*DIGIT "." 1*DIGIT

        String   = DQUOTE *CHAR DQUOTE
                          ; see below for escape characters

        List   = "(" *WSP *1(val *(1*WSP val)) *WSP ")"

        Symbol   = ALPHA *(ALPHA / DIGIT / "_" / "-" /
                          ".")

        Data   = "<" *base64 ">"

    Boolean values are encoded as an integer, with the value of zero
    representing false, and non-zero representing true.

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 different types.

Opaque data is represented as Base64-encoded (see RFC 1521 [7])
character strings surrounded by "< " and "> "

The ABNF syntax definition for Mbus commands is as follows:

```
mbus_command = command_name arglist

command_name = Symbol

arglist      = List
```

Command names SHOULD be constructed hierarchically to group
conceptually related commands under a common hierarchy. The
delimiter between names in the hierarchy MUST be "." (dot).
Application profiles MUST NOT define commands starting with "mbus.".

The Mbus addressing scheme defined in Section 4 allows 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 Section 9.

6. Transport

All messages are transmitted as UDP messages, with two possible
alternatives:

   1. Local multicast/broadcast:
      This transport class MUST be used for all messages that are not
      sent to a fully qualified target address. It MAY also be used for
      messages that are sent to a fully qualified target address. It
      MUST be provided by conforming implementations. See Section 6.1
      for details.

   2. Directed unicast:
      This transport class MAY be used for messages that are sent to a
      fully qualified destination address. It is OPTIONAL and does not
      have to be provided by conforming implementations.

   A fully qualified target address is an Mbus address of an existing
   Mbus entity in an Mbus session. An implementation can identify an
   Mbus address as fully qualified by maintaining a list of known
   entities within an Mbus session. Each known entity has its own
   unique, fully qualified Mbus address.

   Messages are transmitted in UDP datagrams, a maximum message 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 link
   MTU.

   Note that "unicast", "multicast" and "broadcast" mean IP Unicast, IP
   Multicast and IP Broadcast respectively. It is possible to send an
   Mbus message that is addressed to a single entity using IP Multicast.

   This specification deals with both Mbus over UDP/IPv4 and Mbus over
   UDP/IPv6.

6.1 Local Multicast/Broadcast

   In general, the Mbus uses multicast with a limited scope for message
   transport. Two different Mbus multicast scopes are defined, either
   of which can be configured to be used with an Mbus session:

   1. host-local

   2. link-local

   Participants of an Mbus session have to know the multicast address in
   advance -- it cannot be negotiated during the session since it is
   already needed for initial communication between the Mbus entities
   during the bootstrapping phase. It also cannot be allocated prior to
   an Mbus session because there would be no mechanism to announce the
   allocated address to all potential Mbus entities. Therefore, the
   multicast address has to be assigned statically. This document
   defines the use of statically assigned addresses and also provides a

   specification of how an Mbus session can be configured to use non-
   standard, unassigned addresses (see Section 12).

   The following sections specify the use of multicast addresses for
   IPv4 and IPv6.

6.1.1 Mbus multicast groups for IPv4

   For IPv4, a statically assigned, scope-relative multicast address as
   defined by RFC 2365 [11] is used. The offset for the scope relative
   address for Mbus is 8 (MBUS, see
   http://www.iana.org/assignments/multicast-addresses [19]).

   Different scopes are defined by RFC 2365 [11]. The IPv4 Local Scope
   (239.255.0.0/16) is the minimal enclosing scope for administratively
   scoped multicast (as defined by RFC 2365 [11]) and not further
   divisible -- its exact extent is site dependent.

   For the IPv4 Local Scope, applying the rules of RFC 2365 [11] and
   using the assigned offset of 8, the Mbus multicast address is
   therefore 239.255.255.247.

   For IPv4, the different defined Mbus scopes (host-local and link-
   local) are to be realized as follows:

   host-local multicast: Unless configured otherwise, the assigned
      scope-relative Mbus address in the Local Scope (239.255.255.247 as
      of RFC 2365 [11]) MUST be used. Mbus UDP datagrams SHOULD be sent
      with a TTL of 0.

   link-local multicast: Unless configured otherwise, the assigned
      scope-relative Mbus address in the Local Scope (239.255.255.247 as
      of RFC 2365 [11]) MUST be used. Mbus UDP datagrams SHOULD be sent
      with a TTL of 1.

6.1.2 Mbus multicast groups for IPv6

   IPv6 has different address ranges for different multicast scopes and
   distinguishes node local and link local scopes, that are implemented
   as a set of address prefixes for the different address ranges (RFC
   2373 [3]). The link-local prefix is FF02, the node-local prefix is
   FF01. A permanently assigned multicast address will be used for Mbus
   multicast communication, i.e., an address that is independent of the
   scope value and that can be used for all scopes. Implementations for
   IPv6 MUST use the scope-independent address and the appropriate
   prefix for the selected scope. For host-local Mbus communication the
   IPv6 node-local scope prefix MUST be used, for link-local Mbus
   communication the IPv6 link-local scope prefix MUST be used.

The permanent IPv6 multicast address for Mbus/Ipv6 is
FF0X:0:0:0:0:0:0:300.

FF0X:0:0:0:0:0:0:300 SHOULD be used for Mbus/IPv6 where the X in FF0X
indicates that the scope is not fixed because this is an all scope
address. This means, for node-local scope, FF01:0:0:0:0:0:0:300
SHOULD be used and for link-local scope FF02:0:0:0:0:0:0:300 SHOULD
be used. See RFC 2375 [4] for IPv6 multicast address assignments.

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. See Section 12
for specifications of Mbus configuration mechanisms.

6.1.3 Use of Broadcast

In situations where multicast is not available, broadcast MAY be used
instead. In these cases an IP broadcast address for the connected
network SHOULD be used for sending. The node-local broadcast address
for IPv6 is FF01:0:0:0:0:0:0:1, the link-local broadcast address for
IPv6 is FF02:0:0:0:0:0:0:1. For IPv4, the generic broadcast address
(for link-local broadcast) is 255.255.255.255. It is RECOMMENDED
that IPv4-implementations use the generic broadcast address and a TTL
of zero for host-local broadcast.

Broadcast MUST NOT be used in situations where multicast is available
and supported by all systems participating in an Mbus session.

See Section 12 for configuring the use of broadcast.

6.1.4 Mbus UDP Port Number

The registered Mbus UDP port number is 47000.

6.2 Directed Unicast

Directed unicast (via UDP) to the port of a specific application is
an alternative transport class to multicast. Directed unicast is an
OPTIONAL optimization and MAY be used by Mbus implementations for
delivering messages addressed to a single application entity only --
the address of which the Mbus implementation has learned from other
message exchanges before. Note that the DestAddr field of such
messages MUST be filled in properly nevertheless. Every Mbus entity
SHOULD use a single unique endpoint address for sending messages 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 UDP source port number, where the port number is
different from the standard Mbus port number.

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.

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 an 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 a1 is contained in an address element list if
the list contains an element that is equal to a1. An address
element is considered equal to another address element if it has
the same values for both of the two address element fields (tag
and value).

7. 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., to a fully qualified Mbus address. 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
as defined in Section 9.1). A sending entity MUST NOT send a message
reliably if the target address is not unique. (See 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. The desired effect can be achieved at the
application layer by sending individual reliable messages to each
fully qualified destination address, if the membership information
for the Mbus session is available.

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 MUST 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 transmission or the
recipient are considered to have failed. If the message is not
delivered successfully, the sending application is notified. In this
case, it is up to the application to determine the specific actions
(if any) to be taken.

Reliable messages MUST be acknowledged by adding their SeqNum to the AckList field of a message sent to the originator of the reliable message. This message MUST be sent to a fully qualified Mbus target address. Multiple acknowledgments MAY be sent in a single message. Implementations MAY either piggy-back the AckList onto another message sent to the same destination, or MAY 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 MUST transmit the message either via IP-multicast or via IP-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 MUST be retransmitted, the counter N MUST be incremented by one, and the timer MUST 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 MUST be discarded, and the sending application SHOULD be notified.

Receiver: A receiver B of a reliable message from a sender A MUST acknowledge reception 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

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
    continuously 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
    message 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 to the Mbus after startup.
    After transmission of the hello message, it MUST 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
    dependent on the current number of entities in an 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 the number of entities in an Mbus session may vary, care must
be taken to allow the Mbus protocol to automatically scale over a
wide range of group sizes. 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 an
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 implementors 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 an 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 session 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 variable names 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, denoted
   hello_d. The minimum value for hello_d is c_hello_min which yields

      hello_d = max(c_hello_min,c_hello_factor * entities * 1ms).

   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 as follows:

      hello_e = c_hello_dither_min +
                RND * (c_hello_dither_max - c_hello_dither_min)

   with RND being a random function that yields an even distribution
   between 0 and 1. See also 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 an Mbus session a protocol implementation sets
   hello_p=0, hello_now=0 and entities=1, entities_p=1 (the 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 = hello_now +
      (entities/entities_p)*(hello_n - hello_now)

   2. The value for hello_p is updated by setting hello_p = 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.

      1. IF hello_e + hello_p <= hello_now THEN 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 > hello_now THEN 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 Mbus Entities

   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. See Section 8.1.4 for details. 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; these messages are
   required for proper operation of the Mbus. This specification does
   not contain application-specific messages. These are to be defined
   outside of the basic Mbus protocol specification in separate Mbus
   profiles.

9.1 mbus.hello

      Syntax:
      mbus.hello()

      Parameters: - none -

   mbus.hello messages MUST be sent unreliably to all Mbus entities.

   Each Mbus entity learns about other Mbus entities by observing their
   mbus.hello messages and tracking the sender address of each message
   and can thus calculate the current number of entities.

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

   Upon startup the first mbus.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: mbus.bye()

     Parameters: - none -

   An Mbus entity that is about to terminate (or "detach" from the Mbus)
   SHOULD announce this by transmitting an mbus.bye message. The
   mbus.bye message MUST be sent unreliably to all entities.

9.3 mbus.ping

     Syntax: mbus.ping()

     Parameters: - none -

   mbus.ping can be used to solicit other entities to signal their
   existence by replying with an mbus.hello message. Each protocol
   implementation MUST understand mbus.ping and reply with an 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). Several mbus.ping messages MAY be
   answered by a single mbus.hello: if one or more further mbus.ping
   messages are received while the entity is waiting hello_delay time
   units before transmitting the mbus.hello message, no extra mbus.hello
   message need be scheduled for those additional mbus.ping messages.

   As specified in Section 9.1 hello messages MUST be sent unreliably to
   all Mbus entities. This is also the case for replies to ping
   messages. 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 mbus.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 application specific and
  not defined in this document.

  The mbus.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.

  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 else). 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).

  The mbus.waiting message MAY be broadcast to all Mbus entities, MAY
  be multicast to an arbitrary subgroup, or MAY be unicast to a
  particular peer. Transmission of the mbus.waiting message MUST be
  unreliable and hence MUST 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 mbus.waiting messages are sent
  (possibly included in the same Mbus payload). Note that mbus.hello
  and mbus.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 mbus.go message is sent by an Mbus entity to "unblock" another
   Mbus entity -- which has indicated that it is waiting for a certain
   condition to be met. Only a single condition can be specified per
   mbus.go message. If several conditions are satisfied simultaneously
   multiple mbus.go messages MAY be combined in a single Mbus payload.

   The mbus.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                    |     -       |
+------------------+----------------------+-------------+
```

     $T_r = 100ms$

     $N_r = 3$

     $T_c = 70ms$

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

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 11.3). For each
   message, a digest MUST be calculated based on the value of a shared
   secret key value. Receivers of messages MUST check if the sender
   belongs to the same Mbus security domain by re-calculating the digest
   and comparing it to the received value. The messages MUST only be
   processed further if both values are equal. In order to allow
   different simultaneous Mbus sessions at a given scope and to
   compensate defective implementations of host local multicast, message
   authentication MUST be provided by conforming implementations.

   Privacy of Mbus message transport can be achieved by optionally using
   symmetric encryption methods (Section 11.2). Each message MAY 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: the
   algorithms to apply, the keys to use, etc. These and other
   parameters are defined in an Mbus configuration object that is
   accessible by all Mbus entities that participate in an Mbus session.
   In order to achieve interoperability conforming implementations
   SHOULD use the values provided by such an Mbus configuration.
   Section 12 defines the mandatory and optional parameters as well as
   storage procedures for different platforms. Only in cases where none
   of the options mentioned in Section 12 is applicable alternative
   methods of configuring Mbus protocol entities MAY be deployed.

   The algorithms and procedures for applying encryption and
   authentication techniques are specified in the following sections.

11.2 Encryption

   Encryption of messages is OPTIONAL, that means, an Mbus MAY be
   configured not to use encryption.

Implementations can choose between different encryption algorithms. Every conforming implementation MUST provide the AES [18] algorithm. In addition, the following algorithms SHOULD be supported: DES [16], 3DES (triple DES) [16] and IDEA [20].

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 length of the encryption keys is determined by the currently used encryption algorithm. This means, the configured encryption key MUST NOT be shorter than the native key length for the currently configured algorithm.

DES implementations MUST use the DES Cipher Block Chaining (CBC) mode. DES keys (56 bits) MUST be encoded as 8 octets as described in RFC 1423 [12], resulting in 12 Base64-encoded characters. IDEA uses 128-bit keys (24 Base64-encoded characters). AES can use either 128-bit, 192-bit or 256-bit keys. For Mbus encryption using AES only 128-bit keys (24 Base64-encoded characters) MUST be used.

11.3 Message Authentication

For authentication of messages, hashed message authentication codes (HMACs) as described in RFC 2104 [5] are deployed. In general, implementations can choose between a number of digest algorithms. For Mbus authentication, the HMAC algorithm MUST be applied in the following way:

   The keyed hash value is calculated using the HMAC algorithm specified in RFC 2104 [5]. The specific hash algorithm and the secret hash key MUST be obtained from the Mbus configuration (see Section 12).

   The keyed hash values (see RFC 2104 [5]) MUST be truncated to 96 bits (12 octets).

   Subsequently, the resulting 12 octets MUST be Base64-encoded, resulting in 16 Base64-encoded characters (see RFC 1521 [7]).

Either MD5 [15] or SHA-1 [17] SHOULD be used for message authentication codes (MACs). An implementation MAY provide MD5, whereas SHA-1 MUST be implemented.

The length of the hash keys is determined by the selected hashing algorithm. This means, the configured hash key MUST NOT be shorter than the native key length for the currently configured algorithm.

11.4 Procedures for Senders and Receivers

   The algorithms that MUST be provided by implementations are AES and
   SHA-1.

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

   A sender MUST apply the following operations to a message that is to
   be sent:

   1. If encryption is enabled, the message MUST be encrypted using the
      configured algorithm and the configured encryption key. Padding
      (adding extra-characters) for block-ciphers MUST be applied as
      specified in Section 11.2. If encryption is not enabled, the
      message is left unchanged.

   2. Subsequently, a message authentication code (MAC) for the
      (encrypted) message MUST be calculated using the configured HMAC-
      algorithm and the configured hash key.

   3. The MAC MUST then be converted to Base64 encoding, resulting in 16
      Base64-characters as specified in Section 11.3.

   4. At last, the sender MUST construct the final message by placing
      the (encrypted) message after the base64-encoded MAC and a CRLF.
      The ABNF definition for the final message is as follows:

      final_msg = MsgDigest CRLF encr_msg

      MsgDigest = base64

      encr_msg = *OCTET

   A receiver MUST apply the following operations to a message that it
   has received:

   1. Separate the base64-encoded MAC from the (encrypted) message and
      decode the MAC.

   2. Re-calculate the MAC for the message using the configured HMAC-
      algorithm and the configured hash key.

   3. Compare the original MAC with re-calculated MAC. If they differ,
      the message MUST be discarded without further processing.

   4. If encryption is enabled, the message MUST be decrypted using the
      configured algorithm and the configured encryption key. Trailing
      octets with a value of 0 MUST be deleted. If the message does not

start with the string "mbus/" the message MUST be discarded
without further processing.

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 multicast scope to be used for sent messages.

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

The following parameters are optional. When they are present they
MUST be honored. 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.

Use of Broadcast

It can be specified whether broadcast should be used. If
broadcast has been configured implementations SHOULD use the
network broadcast address (as specified in Section 6.1.3)
instead of the standard multicast address.

    Port Number

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

   Two distinct facilities for parameter storage are considered: For
   Unix-like systems a per-user configuration file SHOULD be used and
   for Windows-95/98/NT/2000/XP systems a set of registry entries is
   defined that SHOULD be used. For other systems it is RECOMMENDED
   that the file-based configuration mechanism is 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 RFC 2234 [13]) productions that are later re-used
   for the definitions for the configuration file syntax and registry
   entries:

   algo-id   =   "NOENCR" / "AES" / "DES" / "3DES" / "IDEA" /
                              "HMAC-MD5-96" / "HMAC-SHA1-96"

   scope   =   "HOSTLOCAL" / "LINKLOCAL"

   key   =   base64

   version_number   =   1*10DIGIT

   key_value   =   "(" algo-id "," key ")"

   address   =   IPv4address / IPv6address / "BROADCAST"

   port   =   1*5DIGIT   ; values from 0 through 65535

   Given the definition above, 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 delimiting commas
   MUST always be present though.

   A Base64 string consists of the characters defined in the Base64
   char-set (see RFC 1521 [7]) including all possible padding
   characters, i.e., the length of a Base64-string is always a multiple
   of 4.

   The scope parameter is used to configure an IP-Multicast scope and
   may be set to either "HOSTLOCAL" or "LINKLOCAL". Implementations
   SHOULD choose an appropriate IP-Multicast scope depending on the

value of this parameter and construct an effective IP-Address
considering the specifications of Section 6.1.

The use of broadcast is configured by providing the value "BROADCAST"
for the address field. If broadcast has been configured,
implementations SHOULD use the network broadcast address for the used
IP version instead of the standard multicast address.

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

12.1 File based parameter storage

The file name for an 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 comply to the following syntax definition:

```
    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=" address
```

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 for an 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, Windows XP
registry) SHOULD be used. The hierarchy for Mbus related registry
entries is as follows:

    HKEY_CURRENT_USER\Software\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 | address        |
|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 mechanisms 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 is intended to provide reasonable
security by mandating algorithms and key lengths that are considered
to be cryptographically strong enough at the time of writing.

However, in order to allow for efficiency it is allowable to use
cryptographically weaker algorithms, for example HMAC-MD5 instead of

HMAC-SHA1. Furthermore, encryption can be turned off completely if
privacy is provided by other means or not considered important for a
certain application.

Users of the Mbus should therefore be aware of the selected security
configuration and should check if it meets the security demands for a
given application. Since every implementation MUST provide the
cryptographically strong algorithm it should always be possible to
configure an Mbus in a way that secure communication with
authentication and privacy is ensured.

In any way, 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. When using administratively scoped
multicast, users cannot always assume the presence of correctly
configured boundary routers. In these cases the use of encryption
SHOULD be considered if privacy is desired.

14. IANA Considerations

   The IANA has assigned a scope-relative multicast address with an
   offset of 8 for Mbus/IPv4. The IPv6 permanent multicast address is
   FF0X:0:0:0:0:0:0:300.

   The registered Mbus UDP port number is 47000.

15. References

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

   [2]   Braden, R., "Requirements for Internet Hosts -- Communication
          Layers", STD 3, RFC 1122, October 1989.

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

   [4]   Hinden, R. and S. Deering, "IPv6 Multicast Address
          Assignments", RFC 2375, July 1998.

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

   [6]   Resnick, P., Editor, "Internet Message Format", RFC 2822, April
          2001.

   [7]   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.

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

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

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

   [11] Meyer, D., "Administratively Scoped IP Multicast", BCP 23, RFC
         2365, July 1998.

   [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] Myers, J., "SMTP Service Extension for Authentication", RFC
         2554, March 1999.

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

   [16] U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and
         Technology, "Data Encryption Standard (DES)", FIPS PUB 46-3,
         Category Computer Security, Subcategory Cryptography, October
         1999.

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

   [18] Daemen, J.D. and V.R. Rijmen, "AES Proposal: Rijndael", March
         1999.

   [19] IANA, "Internet Multicast Addresses", URL
         http://www.iana.org/assignments/multicast-addresses, May 2001.

   [20] Schneier, B., "Applied Cryptography", Edition 2, Publisher John
         Wiley & Sons, Inc., status: non-normative, 1996.

Appendix A.   About References

   Please note that the list of references contains normative as well as
   non-normative references. Each Non-normative references is marked as
   "status: non-normative". All unmarked references are normative.

Appendix B.   Limitations and Future Work

   The Mbus is a light-weight local coordination mechanism and
   deliberately not designed for larger scope coordination. It is
   expected to be used on a single node or -- at most -- on a single
   network link.

   Therefore the Mbus protocol does not contain features that would be
   required to qualify it for the use over the global Internet:

      There are no mechanisms to provide congestion control. The issue
      of congestion control is a general problem for multicast
      protocols. The Mbus allows for un-acknowledged messages that are
      sent unreliably, for example as event notifications, from one
      entity to another. Since negative acknowledgements are not
      defined there is no way the sender could realize that it is
      flooding another entity or congesting a low bandwidth network
      segment.

      The reliability mechanism, i.e., the retransmission timers, are
      designed to provide effective, responsive message transport on
      local links but are not suited to cope with larger delays that
      could be introduced from router queues etc.

   Some experiments are currently underway to test the applicability of
   bridges between different distributed Mbus domains without changing
   the basic protocol semantics. Since the use of such bridges should
   be orthogonal to the basic Mbus protocol definitions and since these
   experiments are still work in progress there is no mention of this
   concept in this specification.

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
    3811 N. Fairfax Drive #200
    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

Full Copyright Statement

Acknowledgement