

M3DA Protocol Specification

⊖ This is a draft document

- [M3DA protocol specification](#)
 - [Document history](#)
 - [Reference documents](#)
 - [Table of Content](#)
- [1. Introduction](#)
 - [1.1. Definitions](#)
 - [1.2. Notations](#)
- [2. Serialization](#)
 - [2.1. Object types](#)
 - [2.2. Predefined classes](#)
 - [2.2.1. Transport objects](#)
 - [2.2.1.1. M3DA::Envelope](#)
 - [2.2.2. Message objects](#)
 - [2.2.2.1. M3DA::Message](#)
 - [2.2.2.2. M3DA::Response](#)
 - [2.2.3. Composite objects](#)
 - [2.2.3.1. M3DA::DeltasVector](#)
 - [2.2.3.2. M3DA::QuasiPeriodicVector](#)
- [3. Data model](#)
 - [3.1. Tree structure](#)
 - [3.2. Path and variables](#)
 - [3.3. Shortcuts](#)
 - [3.4. Commands and Events](#)
 - [3.4.1. Events](#)
 - [3.4.2. Commands](#)
- [4. Transport](#)
 - [4.1. Envelope](#)
 - [4.1.1. Nested envelopes](#)
 - [4.1.2. Envelope headers and footers](#)
 - [4.2. Sub M3DA transport](#)
 - [4.2.1. SMS](#)
 - [4.2.1.1. Binary SMS](#)
 - [4.2.1.2. Text SMS](#)
 - [4.2.1.3. GSM SMS format](#)
 - [4.2.2. TCP](#)
 - [4.2.3. UDP](#)
- [5. Work-flow elements](#)
 - [5.1. Package](#)
 - [5.2. Session](#)
 - [5.3. One Way Package](#)
 - [5.4. End of session](#)
- [6. Bysant Custom Object definition](#)

- [7. Examples](#)
 - [7.1. Sequence diagrams](#)
 - [7.1.1. Standard device to server session](#)
 - [7.1.2. Standard device to server to device session](#)
 - [7.1.3. Session on an unreliable with packet loss](#)
 - [7.1.3.1. Case 1: the request envelope is lost](#)
 - [7.1.3.2. Case 2: the reply envelope is lost](#)
 - [7.2. Serialization](#)
 - [7.2.1. Writing a node](#)
 - [7.2.2. Acknowledge a message](#)

M3DA protocol specification

Document history

Date	Version	Author	Comments
Sept 12th 2008	1/A	T. CAMINEL	First released version
March 14th 2009	2	Experts Group	Refactor the document architecture Add correlated data messages
March 24th 2009	2.1	Cuero Bugot David Francois	Add SMS transport Fix typos in <code>M3DA::Delta</code> <code>asVector</code> Precise what an <code>M3DA::Event</code> can hold as data Update serialization examples
March 31st 2009	2.2	Cuero Bugot	Explicit restrictions on ticketIds
April 17th 2009	2.3	Cuero Bugot	Add Binary class in the Classes Diagram Split chapter 4.2 into sub chapters
Sept 18th 2009	2.4	Cuero Bugot	Refactor protocol transport chapter Added Serialization grammar in annexes Add work flows examples Explicit the notion of session

Aug 19th 2011	3.0	Cuero Bugot	Major refactor of the document format (wiki) Removed Timestamped objects Add SMS text transport Removed Events and commands Add shortcut map header field Changed the serialization protocol Add nested envelope Removed HTTP transport
---------------	-----	-------------	--

Reference documents

REF-1	Bysant Serialization Bysant Serializer	Version 1
REF-5	3GPP TS 23.040 Technical realization of the Short Message Service (SMS)	V8.3.0 (2008-09)

Table of Content

- [M3DA protocol specification](#)
 - [Document history](#)
 - [Reference documents](#)
 - [Table of Content](#)
- [1. Introduction](#)
 - [1.1. Definitions](#)
 - [1.2. Notations](#)
- [2. Serialization](#)
 - [2.1. Object types](#)
 - [2.2. Predefined classes](#)
 - [2.2.1. Transport objects](#)
 - [2.2.1.1. M3DA::Envelope](#)
 - [2.2.2. Message objects](#)
 - [2.2.2.1. M3DA::Message](#)
 - [2.2.2.2. M3DA::Response](#)
 - [2.2.3. Composite objects](#)
 - [2.2.3.1. M3DA::DeltasVector](#)
 - [2.2.3.2. M3DA::QuasiPeriodicVector](#)
- [3. Data model](#)
 - [3.1. Tree structure](#)
 - [3.2. Path and variables](#)
 - [3.3. Shortcuts](#)

- [3.4. Commands and Events](#)
 - [3.4.1. Events](#)
 - [3.4.2. Commands](#)
- [4. Transport](#)
 - [4.1. Envelope](#)
 - [4.1.1. Nested envelopes](#)
 - [4.1.2. Envelope headers and footers](#)
 - [4.2. Sub M3DA transport](#)
 - [4.2.1. SMS](#)
 - [4.2.1.1. Binary SMS](#)
 - [4.2.1.2. Text SMS](#)
 - [4.2.1.3. GSM SMS format](#)
 - [4.2.2. TCP](#)
 - [4.2.3. UDP](#)
- [5. Work-flow elements](#)
 - [5.1. Package](#)
 - [5.2. Session](#)
 - [5.3. One Way Package](#)
 - [5.4. End of session](#)
- [6. Bysant Custom Object definition](#)
- [7. Examples](#)
 - [7.1. Sequence diagrams](#)
 - [7.1.1. Standard device to server session](#)
 - [7.1.2. Standard device to server to device session](#)
 - [7.1.3. Session on an unreliable with packet loss](#)
 - [7.1.3.1. Case 1: the request envelope is lost](#)
 - [7.1.3.2. Case 2: the reply envelope is lost](#)
 - [7.2. Serialization](#)
 - [7.2.1. Writing a node](#)
 - [7.2.2. Acknowledge a message](#)

1. Introduction

This document describes the M3DA Protocol stack and exchanged messages.

The document is intended to be read by implementors of the protocol and low level user of the Platform.

The M3DA protocol is a layered stack of protocols and API. At application level there is a set of data structures to access the services of the platform (`M3DA::Message`). At a lower level there is the session and M3DA transport protocol (`M3DA::Envelope`). At the lowest level there are the transports protocols, such as TCP, SMS, etc.

AWTDA

AWTDA Payload

Message

*Command / Event / TimestampedData
/ CorrelatedData*

Message

*Command / Event / TimestampedData
/ CorrelatedData*

...

AWTDA Transport

Headers

*Device / Server Identification
Security*

auth, cipher, hmac, nonce...

Compression

algo, dict selection, ...

Transport

HTTP

TCP

UDP

SMS*

- **Device:** The Device is managed by one remote entities (the Server). A device may have many characteristics, and many parameters may be made available for reading, writing, deleting and modifying by a Device Management Server.
- **Envelope:** The envelope is defined by the protocol object `M3DA::Envelope` object. It is composed of header fields and a payload. The payload is composed of 0, 1 or more messages.
- **Message:** A message is an `M3DA::Message` object holding the basic protocol data elements.
- **One way package:** A one way package is a package that is only composed of a request. This is only used over one way transport layers.
- **Package:** A package is defined as a couple of envelope: a request and a reply. Because of the specific structure of the envelope, one envelope can actually be at the same time a reply of the previous package and a request of the next package.
- **Session:** A Session is a set of one ore more packages happening successively between too peers.

1.2. Notations

`M3DA::` prefix is used to name M3DA custom classes

`B::` prefix is used to name standard Bysant objects

When a pseudo code algorithm is provided:

- arrays start at index 0
- `A.length()` returns the number of elements of the array `A`
- `A.remove(x)` returns and pop the firsts `x` elements of the array `A`

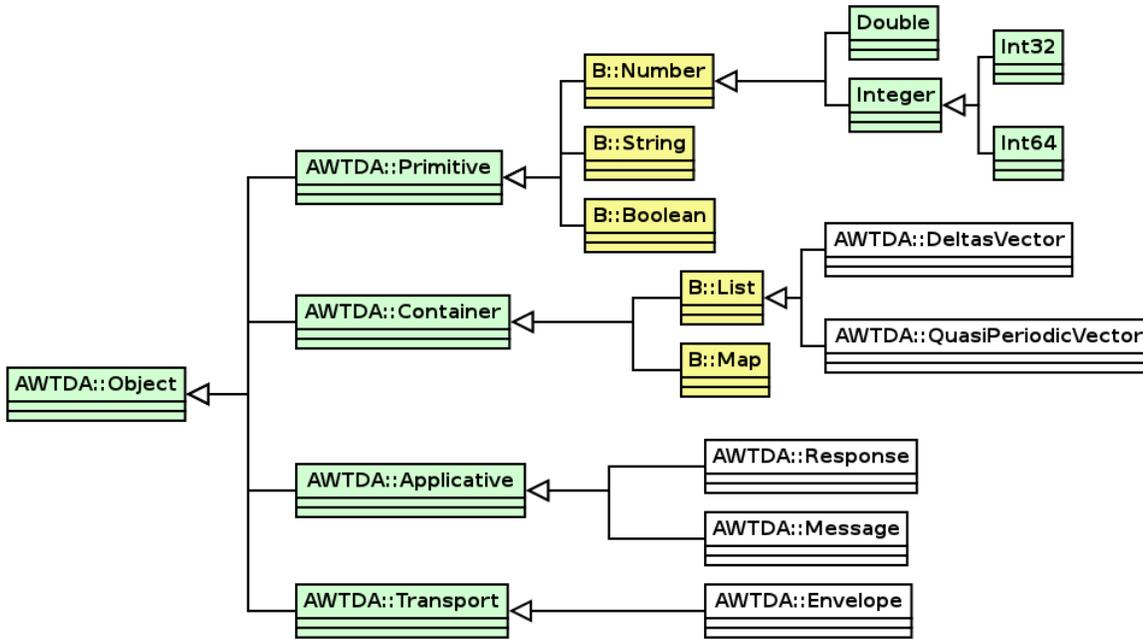
2. Serialization

M3DA uses Bysant serialization [REF-1](#). Bysant is a binary dynamically-typed serialization protocol.

Bysant has been designed to be able to self-describe the serialized types, i.e. not requiring external schema or interface definition. This feature allows flexibility, custom type support. It also eases versioning and extensibility. Bysant has a notion of *reference*, to avoid duplication of type definition. This notion is used to support external object definition, allowing further optimization of the transmitted streams.

2.1. Object types

Bysant uses an internal object model to represent data. It is able to serialize different types of object, including primitive types, basic collections (list/map), and instances of custom classes.



This class diagram represents the hierarchy of classes that are used in this protocol. These classes are a shared model between different implementations that communicate together. The diagram shows a comprehensive view of what can be sent, however this is not an implementation reference. It's rather a view to help implementors to understand how objects will be nested. The only classes that are actually necessary to the application layer are the one in white and yellow (i.e. the final classes)

Implementors must provide at least the final classes (in white/yellow), and can skip generic classes (in green) that are supposed to be abstract.

The primitive classes (in yellow) are not actual subclasses of `M3DA::Object`, but they have been showed this way to explain that primitive type can be contained in objects as standard types. Of course, implementors should rely on their language primitive types to handle this.

2.2. Predefined classes

Bysant allows the reference to an already defined class definition. We use this mechanism to define the classes defined in this document, but the class definition is done out of band. This mean that we have a subset of pre-defined and numbered classes that are shared on each implementation and their definition are never to be sent in any stream.

The following sections describes each predefined classes. Each class has a single unique ID that is used during Bysant serialization: implementations must ensure that this predefined classes are properly processed without any explicit class definitions in the connection streams. To ensure this, note that the given fields are to be serialized in the exact given order. See Bysant specification for more details [REF-1](#).

2.2.1. Transport objects

2.2.1.1. M3DA::Envelope

The `M3DA::Envelope` object is used for the transport of M3DA content. The envelope contains a Header and Footer that store the transport related information, and a payload that contains the M3DA applicative data.

ID	Class Name	Fields
----	------------	--------

0	M3DA::Envelope	<ul style="list-style-type: none"> • header: B::Map • payload: B::String • footer: B::Map
---	----------------	---

The **header** and **footer** maps contains the parameters related to the transport. Having a footer allows to actually stream the data and append data afterwards. This is used for the security extension.

The **payload** is a string of serialized M3DA applicative content (M3DA::Message and M3DA::Response).

 • When no messages are to be sent (for instance for keep alive messages), then an empty string should be serialized.

• When using the security extension, nested envelope may be used. In that case the payload of the first envelope is the second envelope serialized optionally ciphered.

See [#Transport](#) for more details on the envelope.

2.2.2. Message objects

2.2.2.1. M3DA::Message

The M3DA::Message object is a container object that enables the data transmission between two peers

ID	Class Name	Fields
1	M3DA::Message	<ul style="list-style-type: none"> • path: B::String (textual path) or Integer (shortcut) • ticketid: Positive Integer (request ack if not zero) or B::Null (no ack, same as zero) • body: B::Map

The **path** field is the root path of all variables contained in this message.

See chapter [#Path and variables](#) for details on paths. The **path** must not be set to B::Null or empty B::String as it contains the asset id that will receive this message.

The **ticketid** field is used to require an acknowledgment (M3DA::Response) for this message. If no acknowledgment is required then this field must be set to B::Null. The **ticketid** will be repeated in the acknowledgment object.

The sender must ensure that the **ticketid** values do not overlap with a value of a already sent messages but not yet acknowledged, so that received responses can be uniquely matched with one message.

The **body** field is a B::Map containing all the variables names and values. The **body** fields must comply with the following constraints:

- keys
 - the key of a map entry must be either a B::String (textual variable name) or an Integer (shortcut)
 - the key cannot be B::Null or an empty B::String.
 - same rules as the ones for the path definition above apply [#Path and variables](#).
 - it is possible to have composed variable names in the key of the map entry: part of the path can be included in the variable name.
For example, valid keys are: "temperature", "engine.temperature", "input[1]", "engine[1].temperature"

- values
 - the value of a map entry is a B::List of primitive values (B::String, B::Number or B::Boolean).
 - each list of the map must have the same cardinality
 - as a special optimization case, when the B::List holds only one element, the list object can be omitted and replaced by the Value itself
 - lists of values can be actual Bysant B::List object or more specialized list classes as M3DA::DeltasVector or M3DA::QuasiPeriodicVector

2.2.2.2. M3DA::Response

The M3DA::Response object is a container object that enables to acknowledge the processing of a M3DA::Message sent from a peer.

ID	Class Name	Fields
2	M3DA::Response	<ul style="list-style-type: none"> • ticketid: Positive Integer • status: Integer • data: B::String

The **ticketid** field is copied from the M3DA::Message that this response object acknowledges.

The **status** field is a status code that indicate if the processing associated with the corresponding message was successful or not. A value of 0 (zero) means that the corresponding message was processed successfully. A value different from zero means that an error happened.

i It is outside the scope of this specification to define error status codes, or to define when a message was processed successfully or not. This is to be defined at the application level, when implementing the final application.

The **data** field is an additional data along with this response. It is used to describe the error when the status is not equal to 0. This field is optional, and should be set to B::Null when not used.

2.2.3. Composite objects

2.2.3.1. M3DA::DeltasVector

The M3DA::DeltasVector allows improving data compression for data that are similar (data are transmitted as deltas instead of absolute values).

ID	Class Name	Fields
3	M3DA::DeltasVector	<ul style="list-style-type: none"> • factor: B::Number • start: B::Number • deltas: B::List of Integer

This object is a container where numerical values are coded as deltas. A coefficient factor can be applied to minimize further the deltas values.

Actual values Y are recomposed from the object fields in the following manner:

```

Y[0] = factor * start
for i = 0, deltas.length() do
    Y[i+1] = Y[i] + factor * deltas[i]
end

```

Example : the following objects

```

factor=1, start=200, deltas = { 10, -30, 20 }
factor=10, start=20, deltas = {1, -3, 2 }

```

would produce the same set of values:

```
{200, 210, 180, 200}
```

This encoding use less bandwidth, due to the Bysant property to encode small numbers in a more efficient way than big ones.

Moreover, this mechanism allows transmitting data with a given precision. For example, to transmit timestamps with a 1 minute resolution (round the timestamp to a multiple of 60 seconds), the factor should be set to 60.

```

Posix timestamp list : {1233786292, 1233786418, 1233786720, 1233786904 }
Encoding:   factor=60, start=20563105, deltas = { 2, 5, 3 }
Decoding:   {1233786300, 1233786420, 1233786720, 1233786900 }

```

 The list **deltas** only allows integer values in order to save bandwidth

2.2.3.2. M3DA::QuasiPeriodicVector

The M3DA::QuasiPeriodicVector allows improving data compression for data that are quasi periodic (only shifts from a period are sent).

ID	Class Name	Fields
4	M3DA::QuasiPeriodicVector	<ul style="list-style-type: none"> • period: B::Number • start: B::Number • shifts: B::List of B::Number

This object may be used when a vector of values is quasi periodic, meaning they are periodic plus small deltas (due to sampling errors for instance).

Actual values Y are recomposed from the object fields in the following manner:

```

Y[0] = start
local i = 1
while shifts.length() > 1 do
    local n, s = shifts.remove(2)
    for j=i, i+n+1 do
        Y[i] = Y[i-1] + period
    end
    i = i+n+1
    Y[i] = Y[i]+s
end
local n = shifts.remove(1)
for j=i, i+n do
    Y[i] = Y[i-1] + period
end

```

The **shifts** field is never empty and contains an odd number of values ($2k+1$). The first $2k$ values are optional couples (n, s) where n is the number of repeated periods before the shift s is applied; the shifted value is then appended to the list. This process is repeated until all the couples are exhausted. The last value is the number of periods that are repeated to end the list.

Example : the following object

```

period=20, start=143, shifts = { 3, 1, 2, -2, 3}

```

would produce the vector:

```

{143, 163, 183, 203, 224, 244, 264, 282, 302, 322, 342}

```

3. Data model

3.1. Tree structure

The data model is tree oriented.

The root of the tree is the device, the first level is the application, and other levels are optional sub nodes. A variable is a leaf in tree.

An application sub tree may be dynamic or statically defined in an application model.

A node in the data model tree has a unique path.

On each device, a specific application is "@sys". This application is in charge of the Device Management operations.

3.2. Path and variables

When the path or variable name element is a string it should conform to:

- each level of a path should be composed by alphanumeric and " " characters: [0-9a-zA-Z-]

- the path level delimiter is the character "."
- characters "[" and "]" can be used to access lists in the path
- the path level "" (empty string) are removed when processed, meaning:
 - "" concatenated with any path is ignored
 - successive "." in a path are ignored
 - preceding and trailing "." are ignored

All those paths are equivalent: "panel..temperature", ".panel.temperature", "panel.temperature.", the canonical form being "panel.temperature"

Example of valid paths:

- "car.engine[1]"
- "car.engine1" (valid but different meaning from above!)
- "car.battery_level"
- "car"

3.3. Shortcuts

In order to limit bandwidth usage, a shortcut map can be shared between the platform server and the device. The mechanism of keeping this map synchronized is outside the scope of this specification.

In order to use shortcut mechanism, in objects that allows it, a B::String object can be replaced by an Integer object.

Shortcuts are supported for the following fields:

1. Envelope header and footer fields.
2. path field of M3DA::Message object
3. key of the body map field of an M3DA::Message

The shortcut map contains different namespaces for each shortcut categories:

Example:

```

message.keys
{
  0  temperature
  1  voltage
  2  input[1].value
  3  [2]
}

message.path
{
  0  engine
  1  heating.livingroom
  2  @sys.config.monitoring.triggers
}

envelope
{
  0  pid
  1  status
}

```

In order to be sure the shortcut map is synchronized, the sending peer must add the envelope header field `smv` with

the corresponding version of the shortcut map.

If the other peer does not have the definition of the shortcut map for a given version, it should refuse the connection by replying a status code 451. (code value TBD)

3.4. Commands and Events

i This section is **non normative**

M3DA provides an easy way to abstract Command and Event applicative behaviors.

3.4.1. Events

An event is basically a data sending. The receiving peer is free to handle data writing as an event and trigger specific work-flow on data reception.

3.4.2. Commands

A command is a notification plus optional additional parameters. In order to emulate the command behavior, a data writing is to be done on a node path, providing several variables in the message.

For example, the server could send the message containing the path and data:

```
{"@sys.commands.DoSomething", {param1="somevalue", param2="someothervalue"}}
```

The device would receive a notification that the node "@sys.commands.DoSomething" was written, along with the subvalues of param1 and param2

4. Transport

4.1. Envelope

Each exchanges between two peers involve envelopes. An envelope is the object `M3DA::Envelope` that assumes the role of M3DA transport layer. The envelope is an atomic transport element. An envelope is composed of headers, a payload, and footers.

4.1.1. Nested envelopes

Usually the payload of the envelope contains 0, 1 or more messages (`M3DA::Message`); 0, 1 or more responses (`M3DA::Response`). However, specific extensions of M3DA may require that the payload of an envelope contains another envelope with all its fields.

There can be only one nested level. This is used in the M3DA Security extension so to provide protected headers and footers. In that case, the second envelope (the one encapsulated) is called the 'protected envelope'.

4.1.2. Envelope headers and footers

This specification defines headers and footers that should be supported when implementing M3DA. Additional headers and footers may be defined in M3DA extension documents.

Header field name	Description
smv	Shortcut Map Version. This field is mandatory when using the shortcut mechanism. It indicates the shortcut map version used by this package. See chapter TODO for details on shortcuts
id	Sender unique identifier. Used by the other peer to identify the sender. This field is mandatory when the device sends packages to the server, and should be set to the DeviceID.
pid	Package ID. This field is mandatory for unreliable transport. It allows to differentiate consecutive packages from re-emitted packages. The pid value is an integer. See #UDP for details
status	This field is mandatory in reply packages. It gives the status of the previous request.

The following status code are supported:

Status code	Description
200	OK
400	The request is mal-formed.(usually happens when some required header fields are missing or mutually exclusive fields are used together or the envelope content is malformed)
403	This system is not allowed to communicate, authentication will not help.
451	The envelope header contains no reference or a reference to a shortcut map version that does not exist
500	Unexpected server error. The server was able to process the message (e.g. : database outage)
503	Service temporarily unavailable. The request should be retried later. Can happen when the system communication quota is exceeded.

Additional status values can be specified in add-on specifications (e.g. [M3DA Security Extension](#)).

4.2. Sub M3DA transport

Several transport protocols are supported. The M3DA protocol is adaptive to what underlying transport protocol is

used. For this reason `M3DA::Envelope` is self sufficient and no extra features of the transport protocol are used.

4.2.1. SMS

SMS can be used to send small amount of data from/to a modem that support SMS messaging

4.2.1.1. Binary SMS

The binary SMS format is usually the more efficient in terms of data bandwidth.

When using the binary SMS transport protocol, the serialized `M3DA::Envelope` prefixed by the byte string "M3DA" is directly transmitted in the SMS payload.

```
SMS payload = "DA3B" + Serialize({{M3DA::Envelope}})
```

4.2.1.2. Text SMS

Some carriers do not support Binary SMS properly. In that case, text format can be used, but it would be slightly less efficient as the binary data is first translated using base64 algorithm.

When using the text SMS transport protocol, the serialized `M3DA::Envelope` is translated to text using base64 and then prefixed by the byte string "DA3T" and transmitted in the SMS payload.

```
SMS payload = "DA3T" + Base64(Serialize({{M3DA::Envelope}}))
```

4.2.1.3. GSM SMS format

Some other constraints are added on the GSM SMS format:

- Data coding scheme (TP-DCS) must be equal to 4. (Specifying 8-bits uncompressed data)
- Protocol identifier (TP-PID) must equal to 0 (Specifying "Short Message Type 0" SMS)
- If the binary string is larger than the maximum SMS payload size (140 bytes), the standardized "Concatenated SMS" can be used.

The SMS protocol and SMS PDU format is defined in [REF-5](#).

4.2.2. TCP

The TCP transport layer can be used to transport M3DA. The serialized `M3DA::Envelope` object is directly written/read to/from the socket.

Once an M3DA Session is finished, the TCP link can be maintained open in order to save the overhead of reopening another TCP session for a successive M3DA session.

4.2.3. UDP

The UDP transport layer can be used to transport M3DA. The serialized `M3DA::Envelope` object must hold into one single UDP datagram as no fragmentation mechanism is defined over UDP.

Also UDP is an unreliable transport and so the sender must ensure the other peer receives the request by waiting for the reply envelope.

This specification defines a timeout of 5 seconds after which the sent datagram is assumed to be lost and a

re-emission should occur. After 5 failed attempt to send a packet the connection should be aborted and the remote peer considered not reachable.

When using an unreliable transport the envelope header field `pid` is mandatory so to uniquely differentiate packages.

The `pid` header field must be incremented by 1 for each new request and reply envelope, and wraps around at 32. During a session, received UDP datagram must have incremental `pid` (previous `pid` plus one), otherwise the datagram should be discarded.



One device can have only one active connection with a given peer at a given time. That constraint makes sure that received UDP datagrams are only pertaining to the same session.

5. Work-flow elements

5.1. Package

A package is composed of a request and a reply envelope. The request and reply are both formed with the same M3DA protocol object: `M3DA::Envelope`.

- A request envelope is the first envelope of a session or an envelope that have a non empty payload
- A reply envelope is an envelope that contains the `status` field.

During a session, an envelope can serve both as reply (contains the `status` field) and a request (contains some data) at the same time.

It is important to note that only one package at a time is allowed to happen between two peers. In other words, a new request cannot be issued until the reply of the previous request is received.

5.2. Session

A session is composed one or more packages.

5.3. One Way Package

There is an exception for the request-reply scheme when using a one way only transport layer (as SMS for instance). In that case the two peers know that a reply cannot not be sent. The way the two peers know that the package is one-way is outside the scope of this document. It is important to point out the fact that in one way communication a peer cannot know if the message was actually received.

5.4. End of session

A session is finished when an reply-only envelope is received. A reply-only envelope contains the `status` field and no payload.

For a one way package the session correspond exactly to the one way package.

There is also some specific cases where a session should also considered as finished:

1. When the underlying protocol has a session mechanism that encapsulate the M3DA session (like TCP) then the session should be considered finished if the underlying session is finished (ex TCP link is broken).
1. When a peer does not receives a reply for `SESSION_TIMEOUT` (in seconds) time (not considering the retry mechanism that may be used for unreliable links) then the session is considered as finished.

6. Bysant Custom Object definition

M3DA uses Bysant Custom object extension to define M3DA specific objects. This section gives the serialized definition of those objects.

The Bysant class definition is:

```
classID[Numbers] size[Numbers] (ctxID[1 byte]) x size
```

where size is the number of fields of the object. (See [REF-1](#))

M3DA::Envelope: classID[0] size[3] ctxID[6-Lists&Maps] ctxID[1-UIS] ctxID[6-Lists&Maps]

M3DA::Message: classID[1] size[3] ctxID[1-UIS] ctxID[1-UIS] ctxID[6-Lists&Maps]

M3DA::Response: classID[2] size[3] ctxID[1-UIS] ctxID[2-Numbers] ctxID[1-UIS]

M3DA::DeltasVector: classID[3] size[3] ctxID[2-Numbers] ctxID[2-Numbers] ctxID[6-Lists&Maps]

M3DA::QuasiPeriodicVector: classID[4] size[3] ctxID[2-Numbers] ctxID[2-Numbers] ctxID[6-Lists&Maps]

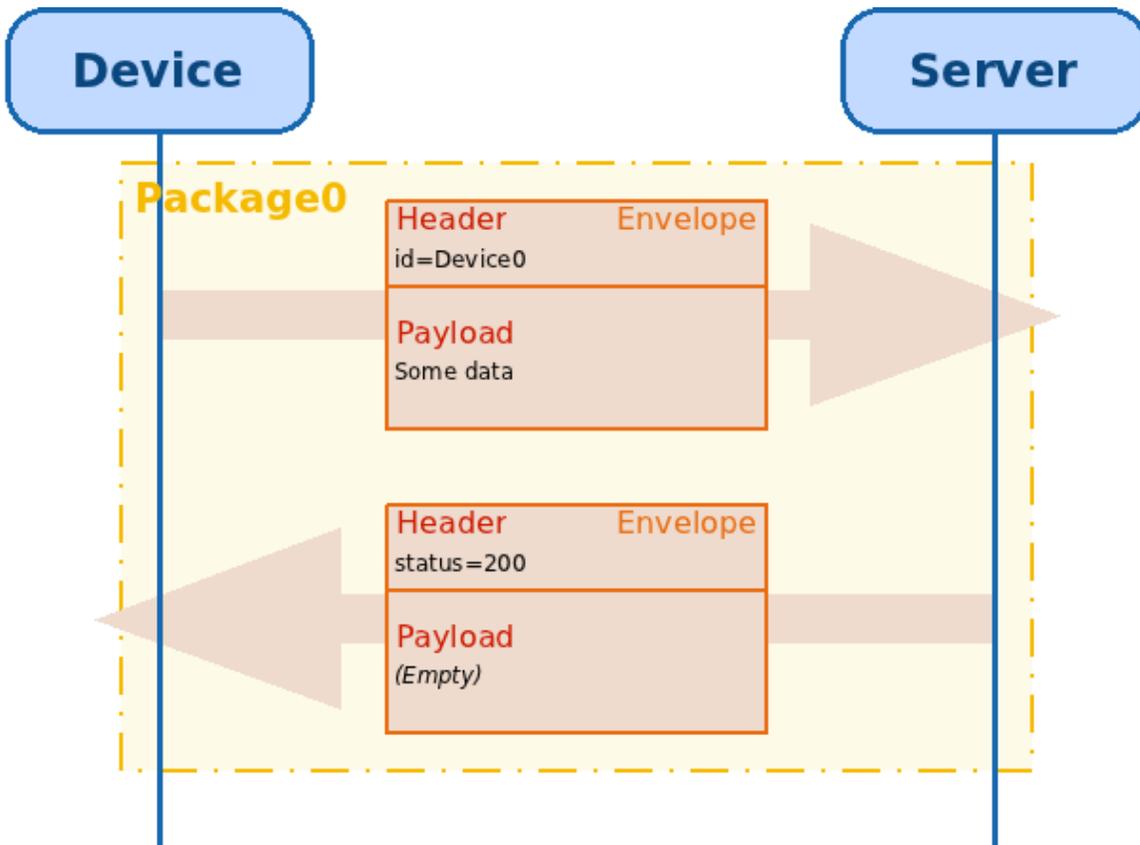
7. Examples

 This section is informative

7.1. Sequence diagrams

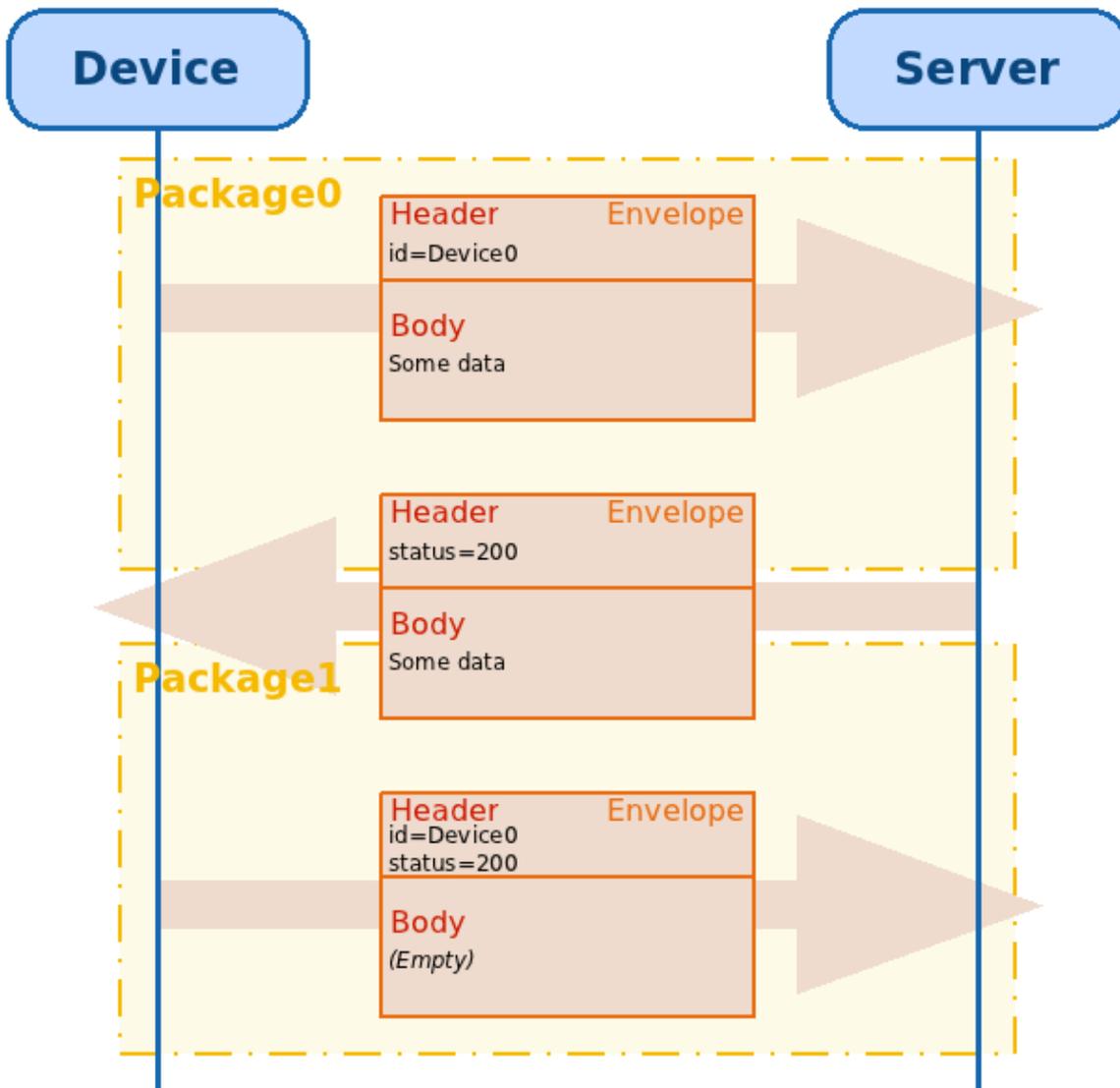
7.1.1. Standard device to server session

In this sequence diagram the device send some data to the server. The server respond with a 200 status empty message, so this finish the current session.



7.1.2. Standard device to server to device session

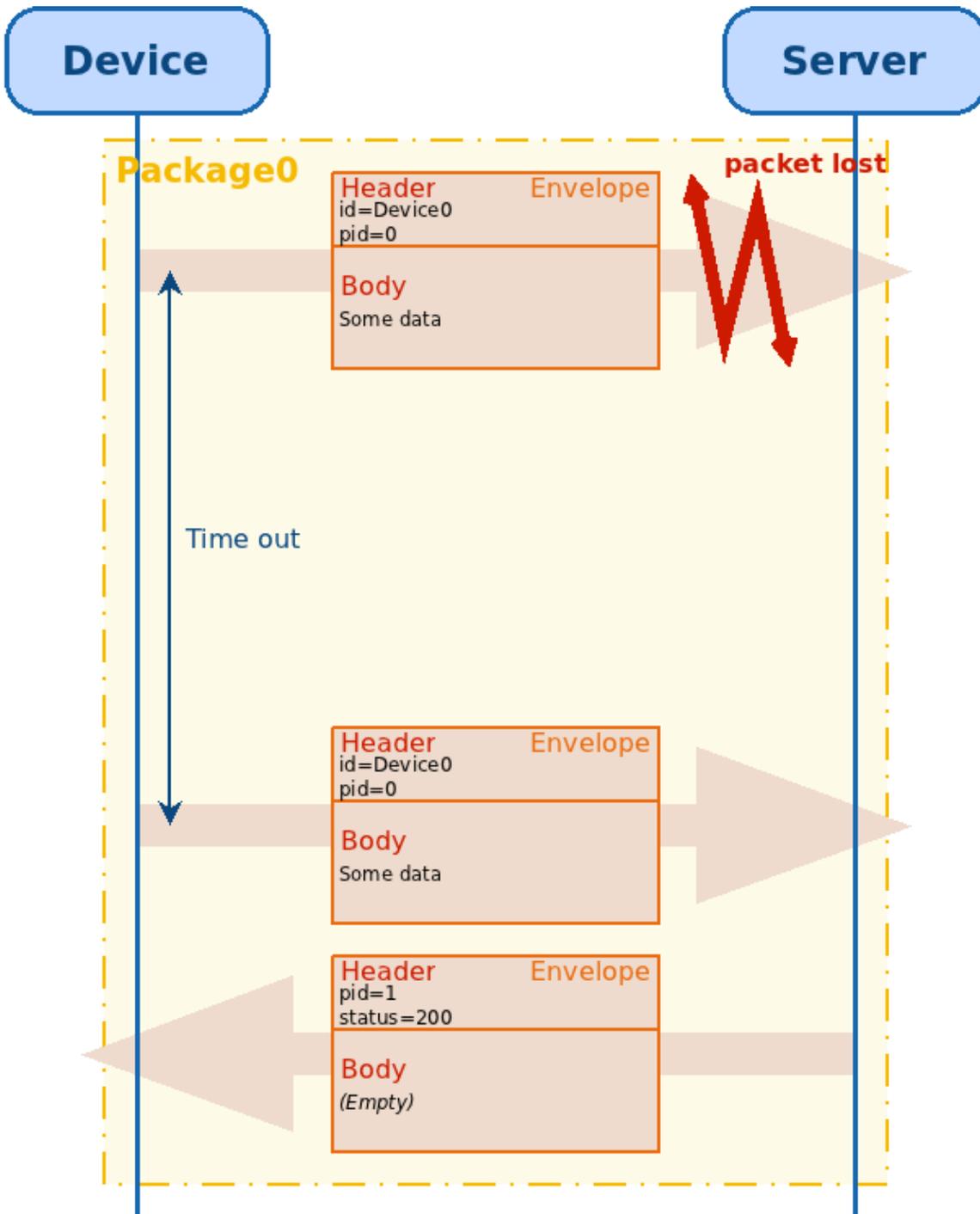
In this sequence diagram the session is done in two packages. The second envelope is actually spanned over two package because it contains both a status code (=>it is a reply) and some data (=>it is a request)



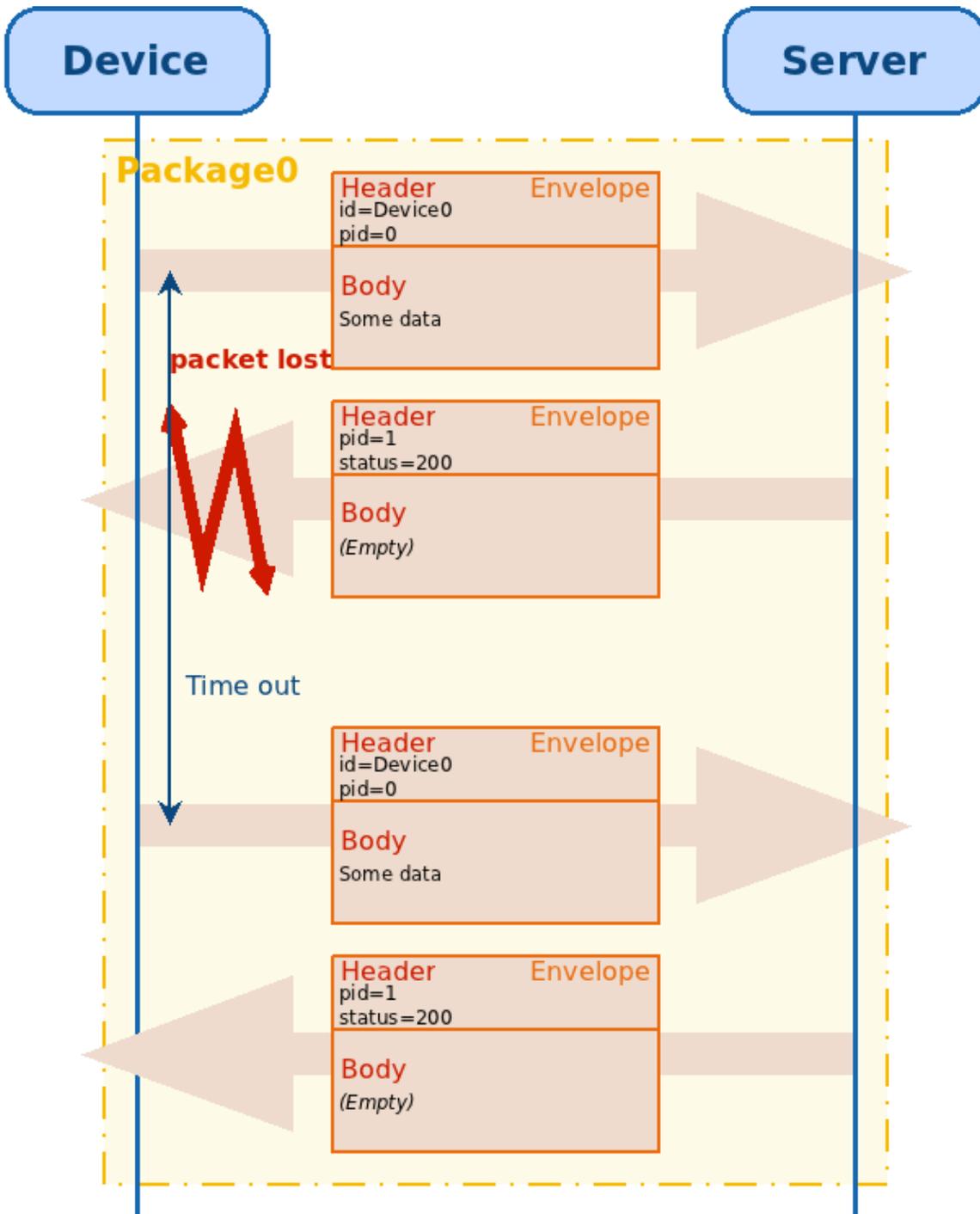
7.1.3. Session on an unreliable with packet loss

In the above sequence diagrams the device re-emit an envelope after a timeout if it has not received the reply envelope. It actually does not matter if the envelope to the server was lost (case 1) or if the reply from the server was lost (case 2), the initial envelope is re-sent. The peer must know what envelope it has already received in case the other peer re-emit it so not to process it twice. This is the purpose of the `pid` field.

7.1.3.1. Case 1: the request envelope is lost



7.1.3.2. Case 2: the reply envelope is lost



7.2. Serialization

7.2.1. Writing a node

TODO

7.2.2. Acknowledge a message

TODO