IEC TS 62325-503

Edition 1.0 2014-01

TECHNICAL SPECIFICATION

colour inside

Framework for energy market communications – Part 503: Market data exchanges guidelines for the IEC 62325-351 profile

IEC TS 62325-503:2014-01(EN)

IEC Central Office
3, rue de Varembé
CH-1211 Geneva 20
Switzerland

Tel.: +41 22 919 02 11
Fax: +41 22 919 03 00
info@iec.ch
www.iec.ch

About the IEC

The International Electrotechnical Commission (IEC) is the leading global organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

About IEC publications

The technical content of IEC publications is kept under constant review by the IEC. Please make sure that you have the latest edition, a corrigenda or an amendment might have been published.

IEC Catalogue - webstore.iec.ch/catalogue

The stand-alone application for consulting the entire bibliographical information on IEC International Standards, Technical Specifications, Technical Reports and other documents. Available for PC, Mac OS, Android Tablets and iPad.

IEC publications search - www.iec.ch/searchpub

The advanced search enables to find IEC publications by a variety of criteria (reference number, text, technical committee,…). It also gives information on projects, replaced and withdrawn publications.

IEC Just Published - webstore.iec.ch/justpublished

Stay up to date on all new IEC publications. Just Published details all new publications released. Available online and also once a month by email.

Electropedia - www.electropedia.org

The world's leading online dictionary of electronic and electrical terms containing more than 30 000 terms and definitions in English and French, with equivalent terms in 14 additional languages. Also known as the International Electrotechnical Vocabulary (IEV) online.

IEC Glossary - std.iec.ch/glossary

More than 55 000 electrotechnical terminology entries in English and French extracted from the Terms and Definitions clause of IEC publications issued since 2002. Some entries have been collected from earlier publications of IEC TC 37, 77, 86 and CISPR.

IEC Customer Service Centre - webstore.iec.ch/csc

If you wish to give us your feedback on this publication or need further assistance, please contact the Customer Service Centre: csc@iec.ch.

IEC TS 62325-503

Edition 1.0 2014-01

TECHNICAL SPECIFICATION

colour inside

Framework for energy market communications – Part 503: Market data exchanges guidelines for the IEC 62325-351 profile

INTERNATIONAL ELECTROTECHNICAL COMMISSION

PRICE CODE XD

ICS 33.200

ISBN 978-2-8322-1368-1

Warning! Make sure that you obtained this publication from an authorized distributor.

CONTENTS

INTERNATIONAL ELECTROTECHNICAL COMMISSION

FRAMEWORK FOR ENERGY MARKET COMMUNICATIONS –

Part 503: Market data exchanges guidelines for the IEC 62325-351 profile

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
5) IEC itself does not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC is not responsible for any services carried out by independent certification bodies.
6) All users should ensure that they have the latest edition of this publication.
7) No liability shall attach to IEC or its directors, employees, servants or agents including individual experts and members of its technical committees and IEC National Committees for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC Publication or any other IEC Publications.
8) Attention is drawn to the Normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.
9) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

The main task of IEC technical committees is to prepare International Standards. In exceptional circumstances, a technical committee may propose the publication of a technical specification when

- the required support cannot be obtained for the publication of an International Standard, despite repeated efforts, or
- the subject is still under technical development or where, for any other reason, there is the future but no immediate possibility of an agreement on an International Standard.

Technical specifications are subject to review within three years of publication to decide whether they can be transformed into International Standards.

IEC/TS 62325-503, which is a technical specification, has been prepared by IEC technical committee 57: Power systems management and associated information exchange.

The text of this technical specification is based on the following documents:

DTS	Report on voting
57/1370/DTS	57/1401/RVC

Full information on the voting for the approval of this technical specification can be found in the report on voting indicated in the above table.

This publication has been drafted in accordance with the ISO/IEC Directives, Part 2.

A list of all parts in the IEC 62325 series, published under the general title *Framework for energy market communications*, can be found on the IEC website.

The committee has decided that the contents of this publication will remain unchanged until the stability date indicated on the IEC web site under "http://webstore.iec.ch" in the data related to the specific publication. At this date, the publication will be

- transformed into an International standard,
- reconfirmed,
- withdrawn,
- replaced by a revised edition, or
- amended.

A bilingual version of this publication may be issued at a later date.

IMPORTANT – The 'colour inside' logo on the cover page of this publication indicates that it contains colours which are considered to be useful for the correct understanding of its contents. Users should therefore print this document using a colour printer.

INTRODUCTION

This Technical Specification is part of the IEC 62325 series which defines protocols for deregulated energy market communications.

The principal objective of the IEC 62325 series is to produce standards which facilitate the integration of market application software developed independently by different vendors into a market management system, between market management systems and market participant systems. This is accomplished by defining message exchanges to enable these applications or systems access to public data and exchange information independent of how such information is represented internally.

The common information model (CIM) specifies the basis for the semantics for the message exchange. The European style market profile specifications that support the European style design electricity markets are defined in IEC 62325-351. These electricity markets are based on the European regulations, and on the concepts of third party access and zonal markets. The IEC 62325-451-n International standards specify the content of the messages exchanged.

The purpose of this technical specification is to provide the guidelines to exchange the above mentioned messages. A European market participant (trader, distribution utilities, etc.) could benefit from a single, common, harmonized and secure platform for message exchange with the European Transmission System Operators (TSOs); thus reducing the cost of building different IT platforms to interface with all the parties involved.

This Technical Specification represents an important step in facilitating parties entering into electricity markets other than their national ones; they could use the same or similar information exchange system to participate in more than one market all over Europe.

This Technical Specification was originally based upon the work of the European Network of Transmission System Operators (ENTSO-E) Working Group EDI.

FRAMEWORK FOR ENERGY MARKET COMMUNICATIONS –

Part 503: Market data exchanges guidelines for the IEC 62325-351 profile

1 Scope

This technical specification is for European electricity markets.

This document specifies a standard for a communication platform which every Transmission System Operator (TSO) in Europe may use to reliably and securely exchange documents for the energy market. Consequently a European market participant (trader, distribution utilities, etc.) could benefit from a single, common, harmonized and secure platform for message exchange with the different TSOs; thus reducing the cost of building different IT platforms to interface with all the parties involved. This also represents an important step in facilitating parties entering into markets other than their national ones.

From now on the acronym “MADES” (MArket Data ExchangeS) will be used to designate these Technical Specifications.

MADES is a specification for a decentralized common communication platform based on international IT protocol standards:

- From a business application (BA) perspective, MADES specifies software interfaces to exchange electronic documents with other BAs. Such interfaces mainly provide means to send and receive documents using a so-called “MADES network”. Every step of the delivery process is acknowledged, and the sender can request about the delivery status of a document. This is done through acknowledgement, which are messages returned back to the sender. This makes MADES networks usable for exchanging documents in business processes requiring reliable delivery.
- MADES also specifies all services for the business application (BA); the complexities of recipient localisation, recipient connection status, message routing and security are hidden from the connecting BA. MADES services include directory, authentication, encryption, signing, message tracking, message logging and temporary message storage.

The purpose of MADES is to create a data exchange standard comprised of standard protocols and utilizing IT best practices to create a mechanism for exchanging data over any TCP/IP communication network, in order to facilitate business to business information exchanges as described in IEC 62325-351 and the IEC 62325-451 series.

A MADES network acts as a post-office organization. The transported object is a “message” in which the sender document is securely repackaged in an envelope (i.e. a header) containing all the necessary information for tracking, transportation and delivery.

2 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEC 61970-2, *Energy management system application program interface (EMS-API) – Part 2: Glossary*

IETF RFC 1738, *Uniform resource locators (URL), http://www.ietf.org/rfc/rfc1738.txt*

IETF RFC 3110, *RSA/SHA-1 SIGs and RSA KEYs in the domain name system (DNS),http://www.ietf.org/rfc/rfc3110.txt*

IETF RFC 4122, *A universally unique identifier (UUID) URN namespace, http://www.ietf.org/rfc/rfc4122.txt*

ITU-T Recommendation X.509, *Information technology - Open systems interconnection - The directory: Public-key and attribute certificate frameworks, http://www.itu.int/rec/T-REC-X.509/en*

3 Terms and definitions

For the purposes of this document, the terms and definitions given in IEC 61970-2 apply, as well as the following.

NOTE General glossary definitions can be found in IEC 60050, *International Electrotechnical Vocabulary*.

3.1
advanced encryption standard
AES
symmetric cryptographic algorithm

3.2
distinguish encoding rule
DER
format for X.509 digital certificates

3.3
European style market profile
ESMP
European style market profile for which this Technical Specification is designed

3.4
market data exchange standard
MADES
standard described in this document for the European market style market profile

3.5
profile
basic outline of all the information that is required to satisfy a specific environment

3.6
transmission system operator
TSO
company responsible for operating, maintaining and developing the transmission system for a control area and its interconnections

4 High level concepts

4.1 What is MADES intended for?

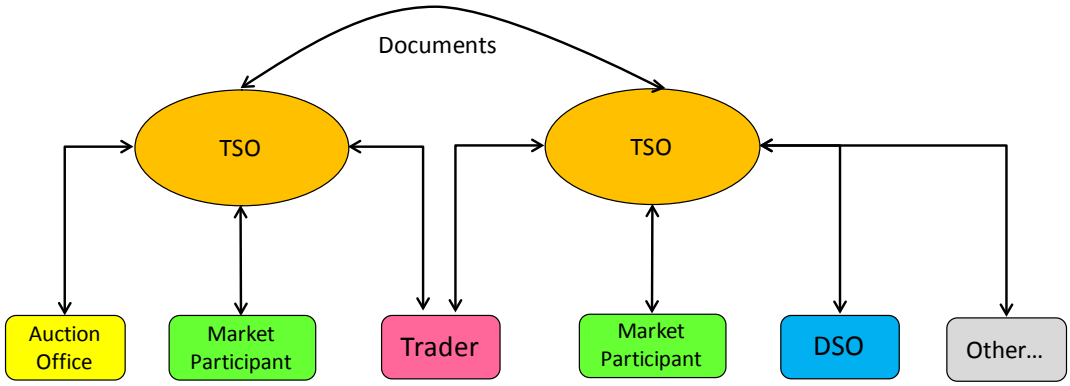

Figure 1 – MADES overall view

MADES' first intention is to provide TSOs with a standardized communication access point to securely exchange documents with others parties involved in the European electricity market as shown in Figure 1. These documents are mainly the ones used in the energy market and described in IEC 62325-351 and the IEC 62325-451 series. Such parties include TSOs, distribution system operators (DSO), balance responsible parties (BRP), capacity traders (CT), market operators (MO), producers, transmission capacity allocators (TCA), etc.

The MADES enables each party to implement MADES access points (referred to as endpoints) connected to his information system (IS), where he may securely send and receive documents to and from other parties.

MADES is a market data exchange standard comprised of standard protocols and utilizing IT best practices to create a mechanism for exchanging data over any TCP/IP communication network, in order to facilitate business-to-business information exchanges.

New market rules induce new business processes and activities, and generally require new information exchanges between parties. Experience shows that, for the exchanges to operate according to the business goals, the chosen technical solution results from an agreement of involved parties gathering various constraints, including implementation time scale, vendors' offer, already existing communication links, integration capabilities of existing information systems, confidentiality of exchanged information, legal risks, etc.

Where business processes require information to be exchanged between multiple systems or multiple parties, solutions developed bilaterally may become extremely complex, with each interface taking time, money and resources to be developed and be maintained. It is also a noticeable consequence that some parties acting in several countries, such as traders, may have to install different communication tools in order to interface with different trading solutions. The future vision is a single interface between all parties in all areas of the electricity market of Europe.

MADES can support any business process whatever the document types being transmitted might be (e.g. XML, binary) and whatever the sequence for the exchanges.

MADES is independent of the physical underlying communication Infrastructure, which can be any IP (Internet Protocol) network, such as Internet, a physical private infrastructure, or a multi access-point virtual private network (VPN).

MADES relies on and only on non-proprietary IT standards for communication protocols, data integrity, signing and confidentiality (encryption), peer access point authentication, peer party authentication, parties' directory (e.g. HTTPS, SOAP, X.509), as shown in Figure 2.

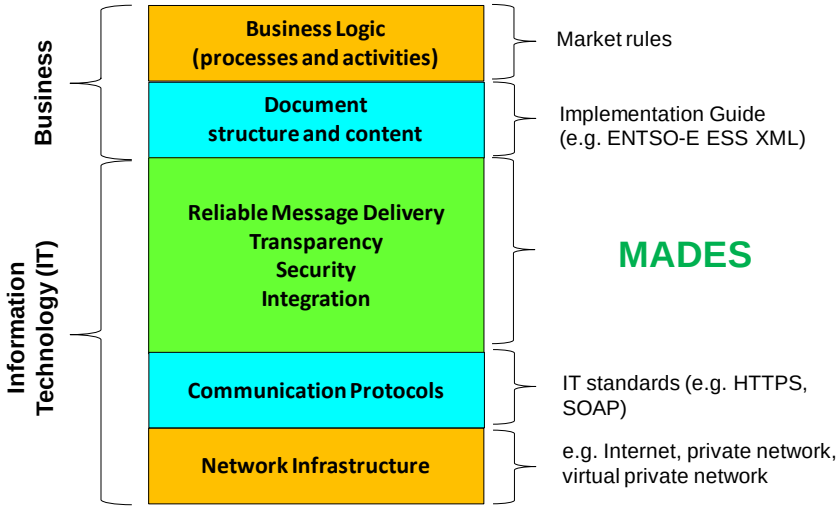

Figure 2 – MADES scope

4.2 General overview

The purpose of the MADES standard is to specify a message delivery platform with the key features shown in Figure 3.

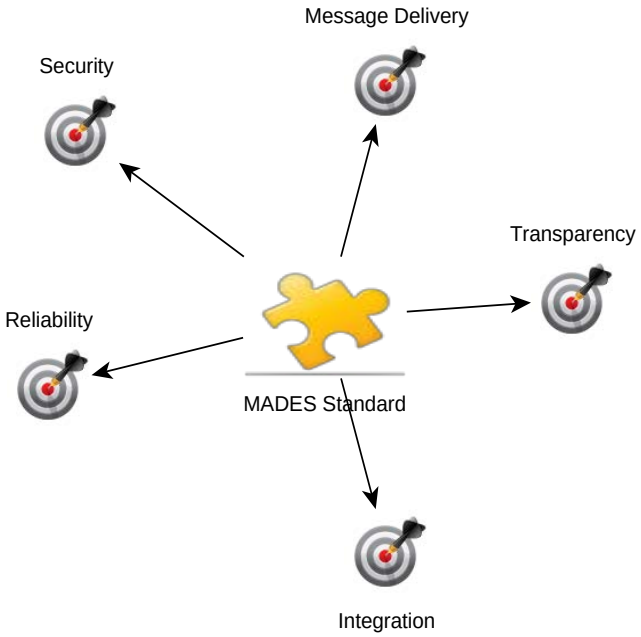

Figure 3 – MADES key features

1. **Message delivery** – A party (sender) connected to the communication network can send a message to another party (recipient), which is connected or can connect to the network.
2. **Transparency** – Any transported message can be tracked down to gather trustworthy information about the state of delivery and traversal path.

3. **Security** – Only the recipient of the message is capable of reading the message-content. The sender of any message can be unambiguously verified.
4. **Reliability** – A message cannot get lost.
5. **Integration** – The MADES functions for sending and receiving messages can be integrated with wide variety of technologies.

The first four key features (message delivery, security, transparency and reliability) are capabilities of the communication system, while the other one (integration) is a design characteristic of the components of the communication system.

4.3 Message delivery and transparency

4.3.1 Message delivery

The main feature of MADES is the message delivery function, as shown Figure 4.

Figure 4 – MADES message delivery overview

A message is transferred from a sender to a recipient. Both sender and recipient are business applications (BAs). A BA connects to a MADES endpoint using a programming interface.

The sender and recipient view the MADES system only through the defined interface. The document transported between sender and recipient can be any text or binary data. Alongside with the document, a MADES message contains additional information, in a header (or envelope), including information to securely identify, transport and route the message such as a unique message ID, the identities of the sender and of the recipient, a business-type.

4.3.2 Transparency

The message path – from the sender's endpoint to the recipient's endpoint – goes through some components of the MADES network. When a message traverses a component, the later notifies the event and a new message (referred as an acknowledgement) is sent back to the sender's endpoint. All the events notified during the message delivery can be retrieved by the sender's BA.

4.4 Security and reliability

MADES ensures a secure message transfer and a fully tracked delivery, as shown in Figure 5.

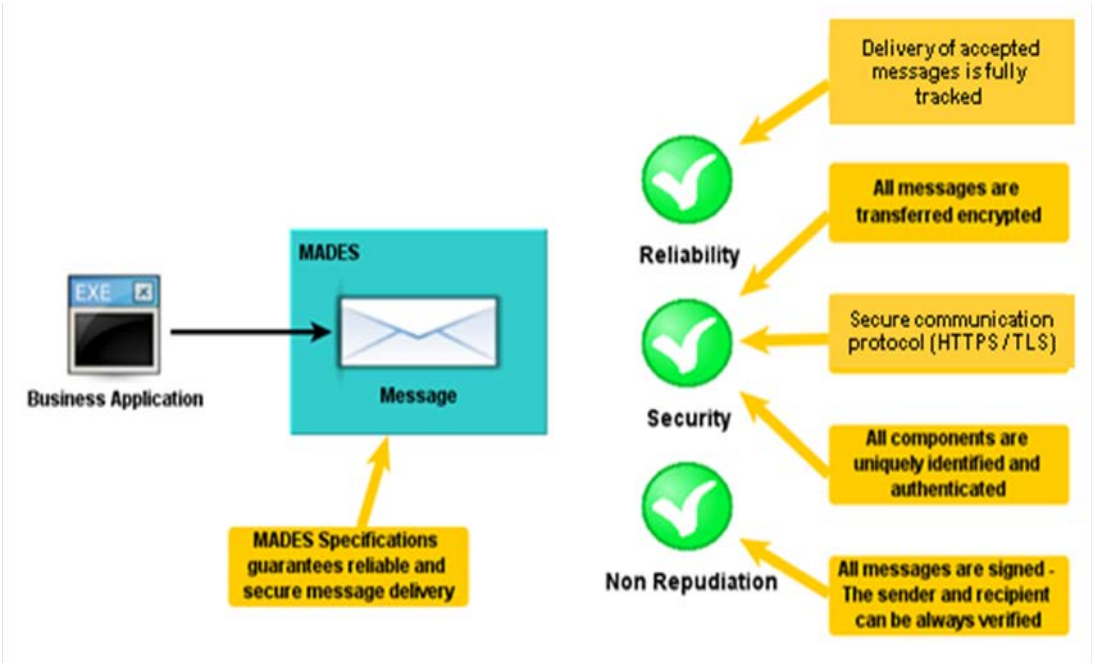

Figure 5 – MADES security and reliability

A MADES communication system guarantees that any message accepted by the system will not be lost. The sender can at any time check the delivery status of the messages (delivering, delivered or failed).

The standard describes a logging mechanism to be implemented in all message handling components to provide information about the message transfers; MADES describes non-repudiation features, allowing the verification of a message and its header which includes the sender, the recipient, the sending time, the delivery time, etc.

MADES defines the way to sign and encrypt the transported messages.

For the communication layer, the MADES components use the secure communication protocols HTTPS. Information is transported encrypted. Moreover, both sides of communication are authenticated using industry-standard PKI certificates.

The security features are detailed in 4.8.

4.5 Main components

MADES describes three logical communication components and their interfaces, as shown in Figure 6.

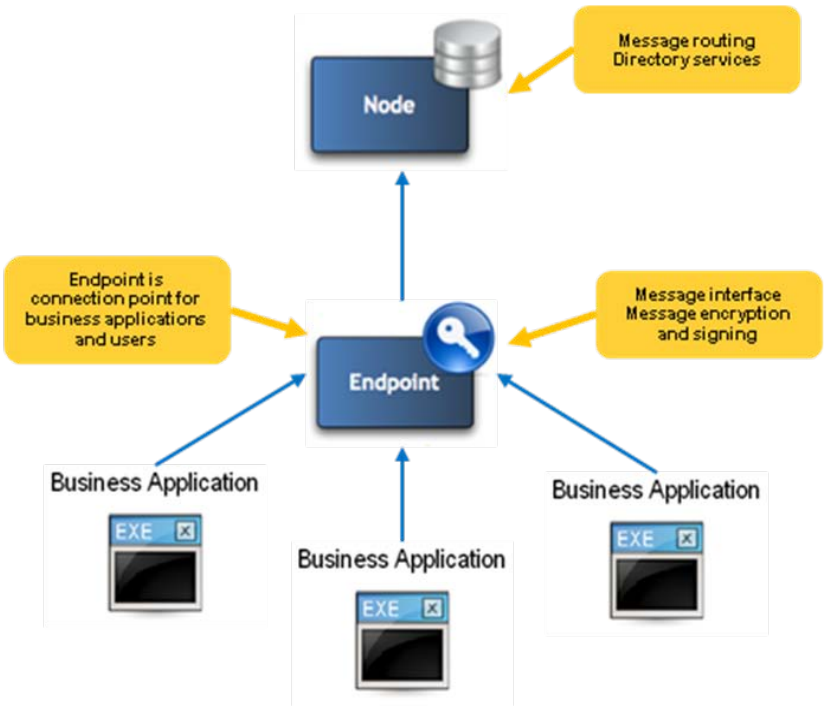

Figure 6 – MADES components

From the users' or business application (BA) point of view, the crucial component is the endpoint, which provides the interface for the BAs to send and receive the messages. Actually no graphical user interface is part of MADES; such an interface to provide a manual way for sending and receiving messages is an application which can be integrated with the endpoint.

The node component serves as a central part of a MADES network. Each node contains a directory with information on all the registered network components, whether endpoints or nodes.

4.6 Distributed architecture

A MADES network may consist of multiple interconnected nodes, each taking care of a part of the network, as shown in Figure 7.

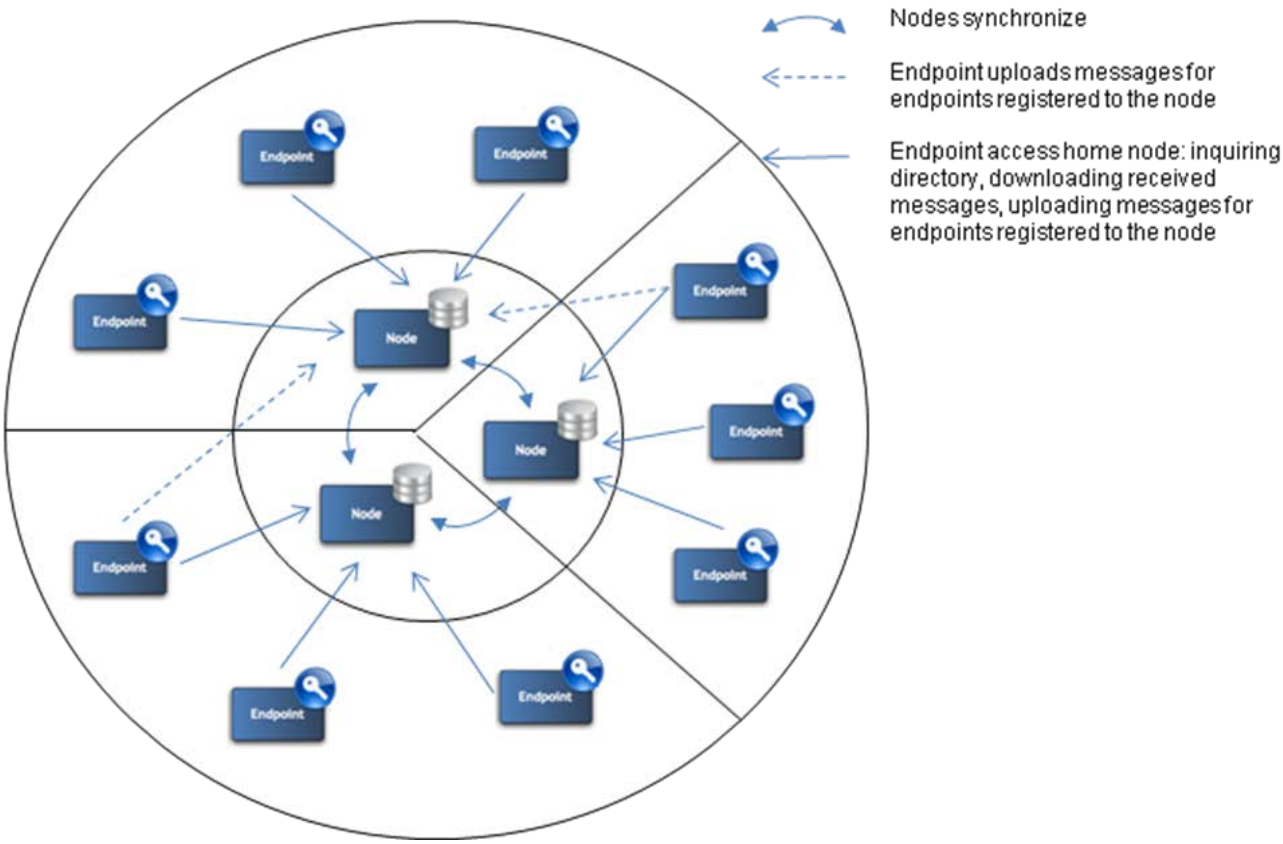

Figure 7 – MADES network distributed architecture

A MADES network may contain a large number of collaborating components with the nodes in the centre. A MADES network has a distributed architecture; it does not have any single central component. All nodes have equal responsibilities; each manages a part of the network.

Each endpoint shall register with a home node. The components registered with a node are referred as the registered components. Endpoints currently connected to a node are the connected endpoints.

Directory information about all registered endpoints is regularly shared between the nodes, using the node synchronization interface; so that endpoints registered with different home nodes can exchange messages.

An endpoint can connect to any node to send messages, but it can only receive messages from the home node.

4.7 Components' exposed interfaces

MADES standard specifies the interfaces between the components. All interfaces and services are presented in Figure 8.

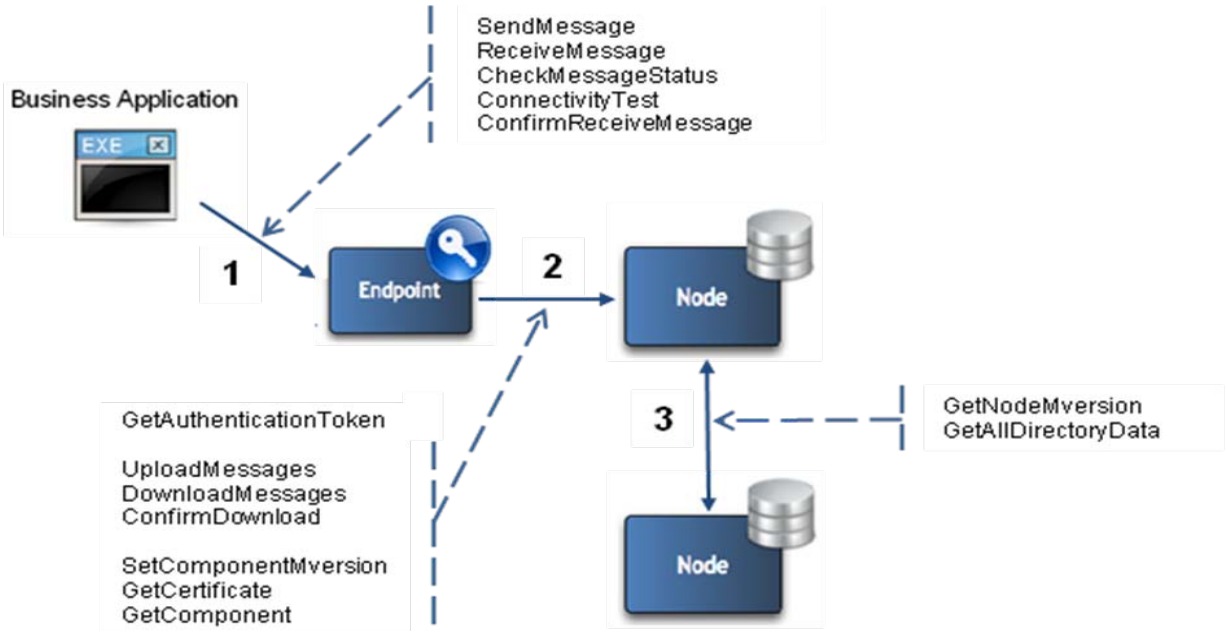

Figure 8 – MADES interfaces and services

Each arrow on Figure 8 shows a component (at the tail of the arrow) using the interface exposed by the component at the tip of the arrow.

a) Endpoint interface → used by a business application (BA) – see 7.2.

b) An endpoint shall implement this interface for a BA to connect to a MADES network.

c) Node interface → used by the endpoints – see 7.3.

d) A node shall implement this interface to allow an endpoint to transfer the messages and to query the node directory.

e) Node synchronization interface → used by the nodes – see 7.3.5.

f) A node shall implement this interface to synchronize directory data with the other nodes of the MADES network.

4.8 Security features

4.8.1 Overview

Main goals of the MADES security definition are summarized by the following points:

- The security solution is transparent to the business applications (BAs) – no specific implementation shall be required in the application to communicate securely.
- Any message shall be readable only by the recipient.
- The sender of any message shall be unambiguously identified.
- Non-repudiation of the messages – it shall be possible to unambiguously prove that the sender sent the message and that the recipient received it.
- All communication routes shall be encrypted. The security solution complies with the X.509 public key infrastructure.

The security issues are covered on two levels: transport-layer security and message-level security.

On the transport-layer, MADES requires the communication between two components to always be encrypted. Two components that exchange information shall first unambiguously identify each other.

On the message-level, MADES requires that all messages shall be signed and encrypted, so the sender of the message can be unambiguously identified and the message is only readable by the intended recipient.

4.8.2 Transport-layer security

The transport–layer security of the communication between components relies on the transport protocol layer. When communicating, components shall use a secure protocol HTTPS providing the encryption of the communication route. Mutual authentication of communicating components shall be handled using X.509 certificates; both the client and the server shall authenticate themselves by their respective authentication X.509 certificates, as shown in Figure 9.

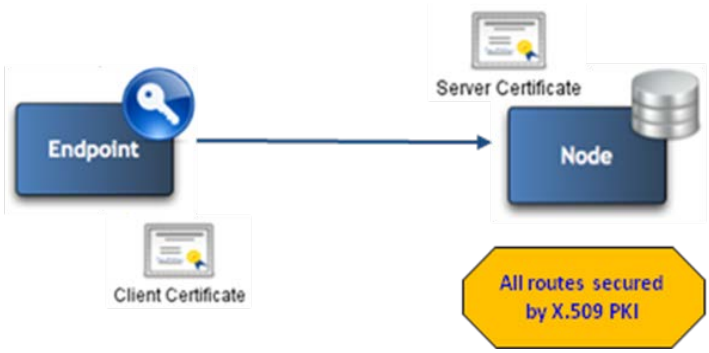

Figure 9 – MADES transport security overview

The communication (i.e. the IP connection) between components shall always be initiated by the client. This provides higher security on client-side by not having to allow incoming connections through firewalls, as shown in Figure 10.

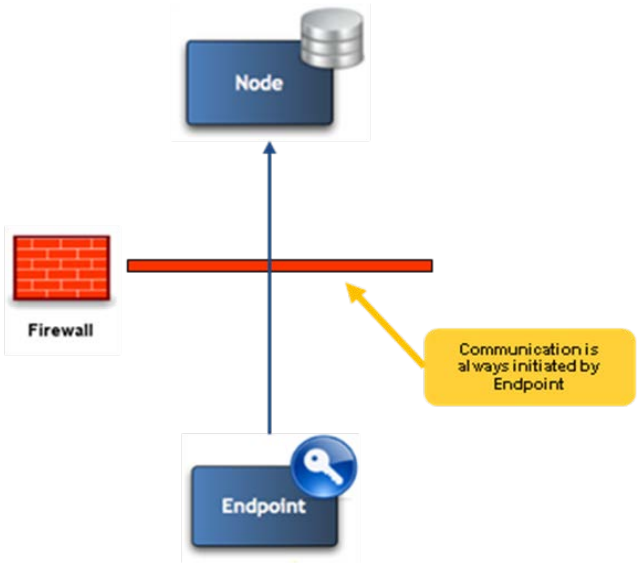

Figure 10 – MADES secure communication initiation

4.8.3 Message-level security

The unambiguous identification of the sender of any message sent via the MADES network is enabled by usage of digital signatures as shown in Figure 11.

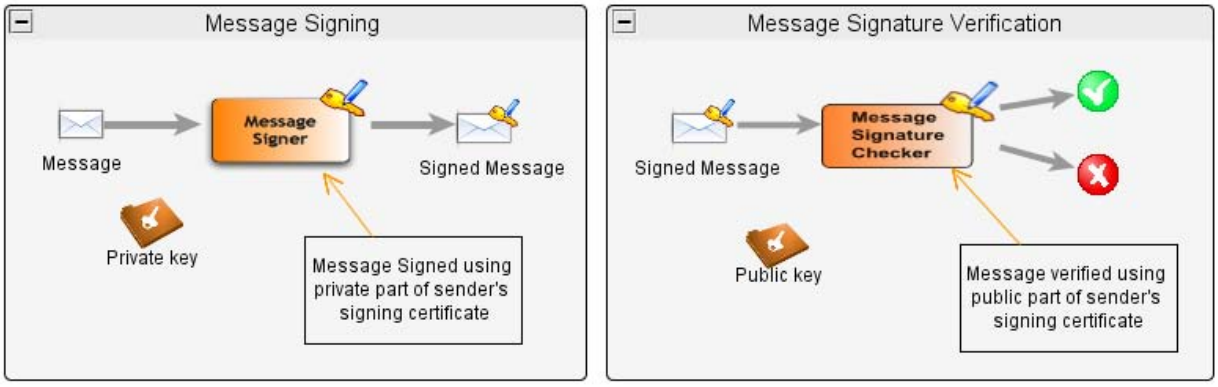

Figure 11 – Message signature

On the sender's endpoint, the message is signed using the sender's private key of a signing certificate. On the recipient's endpoint, the message signature is verified using the sender's public key of the certificate.

Any message sent via the MADES shall be encrypted, so that only the intended recipient can read the message-content as shown in Figure 12.

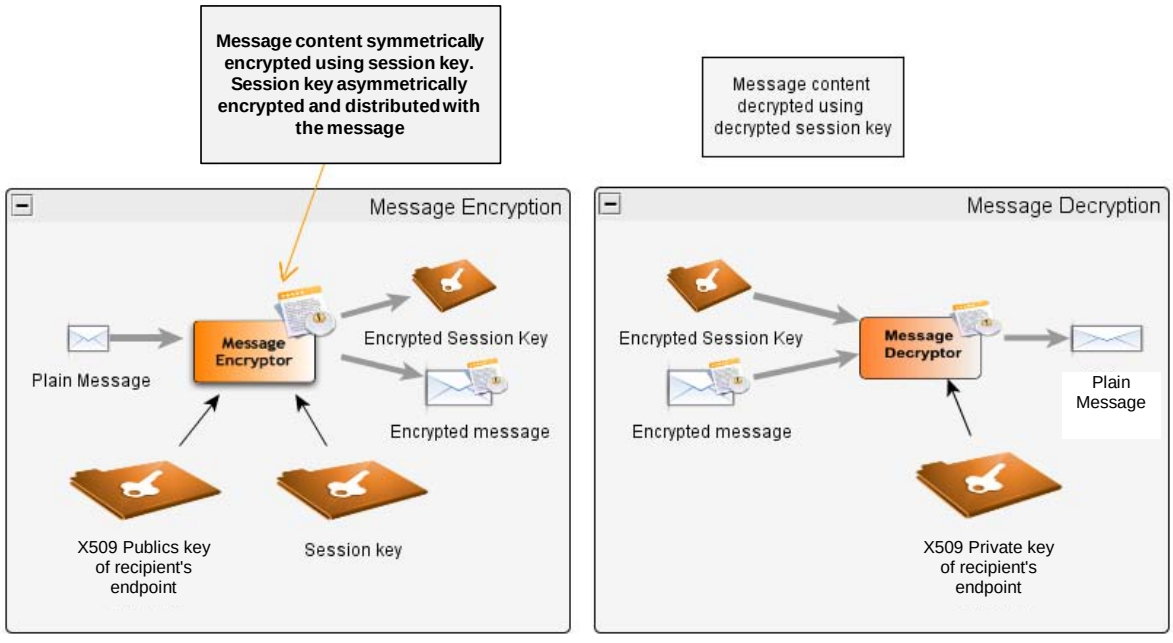

Figure 12 – Message encryption and decryption

The content of the message (i.e. the document) is encoded with a randomly generated session key, which is then itself encoded with the public key of the encryption certificate of the recipient. The encoded key is transported together with the message.

The receiver decodes the key with the private key of his encryption certificate, and then uses the key to decode the document.

4.8.4 Non repudiation

4.8.4.1 Overview

The general behaviour is first presented in Figure 13, and then detailed in 4.8.4.2 and 4.8.3.

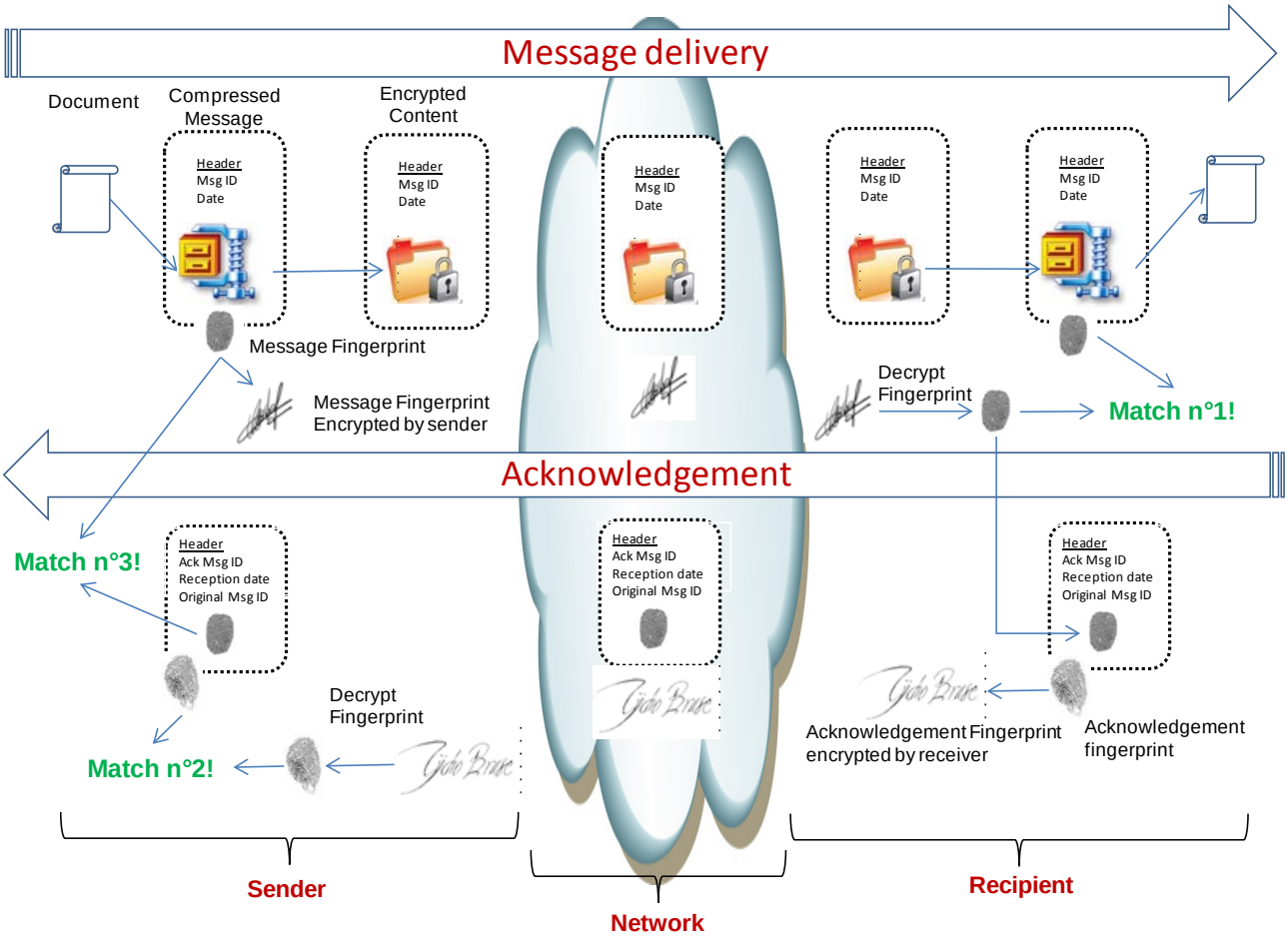

Figure 13 – Non repudiation

4.8.4.2 Message delivery

The message fingerprint uniquely identifies the document together with some header information, such as the message unique ID (MsgID) and the sending date and time. The document may have been previously compressed.

Both the fingerprint and the document are encoded and transported to the recipient. The fingerprint is encoded in such a way that uniquely identifies the sender (signature), and the document in such a way that only the recipient can read it (encryption).

The recipient decodes both the fingerprint and the document. He verifies (match n°1) that the fingerprint, which he can regenerate from the message, matches the transported signed fingerprint (signature verification). Then he stores the decoded message and the encoded fingerprint. Both elements together with the signing certificate prove that the message was sent by the sender.

4.8.4.3 Acknowledgement

The recipient sends back a new message, the acknowledgement, using a similar process. The new message contains the unique ID of the original message (Original Msg ID) and the attached document is the fingerprint of the original message[1].

The acknowledgement is signed but not encrypted and transported to the sender of the original document.

When he receives the acknowledgement, the sender verifies (match n°2) that the acknowledgement was sent by the recipient (signature verification). He also verifies that the acknowledgement document is the original message fingerprint (match n°3).

The set, composed of the original message, the signed acknowledgement and the signing certificate, proves that the recipient received the original message.

5 Components' functions

5.1 Routing messages

A message shall be transported from the sender's BA (business application) to the recipient's BA using the route shown in Figure 14.

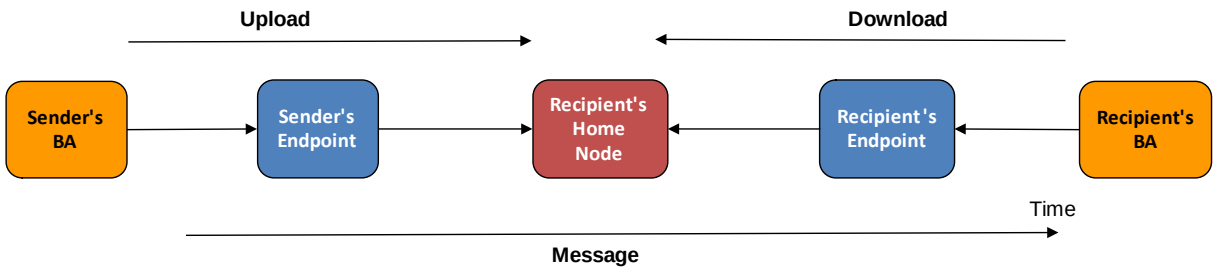

Figure 14 – Delivery route of a business-message

The message is composed by the sender's endpoint with information and document provided by the sender's BA. Then the message is transferred from component to component (from left to right in Figure 14) until the recipient's BA.

The message-content (also referred as the message-payload) is the document provided by the sender's BA. The composed message contains additional information (i.e. a header) used for security, routing and delivery tracking.

The arrows in Figure 14 represent the IP connections between the components and each arrow goes from a client to a server. Thus, a message is uploaded (or sent, or going out) on the way from sender's BA to the recipient's node. It is downloaded (or received, or coming in) on the way from the node to the recipient's BA. "Transfer" is the generic word used to mean either upload or download.

The node, which the message goes through, shall be the home node of the recipient's endpoint. Note that a reverse going message between the two BAs will not use the reverse route if sender and recipient have different home nodes.

There are two types of messages:

---

1 There are several acknowledgements during the message delivery; the one referred to here is sent by the endpoint of the recipient party after it correctly receives the message (event n°4 in Figure 19).

- Business-message – is a message composed by the sender's endpoint from a send request initiated by a sender's BA. The goal of MADES is to transport such business-messages to the requested recipient's endpoint.
- Acknowledgement – is an ancillary message used for tracking the end-to-end delivery process of a business-message. The BAs do not know about those internal acknowledgements, but a BA can request the endpoint about the delivery status of a previously sent business-message.

5.2 Component and message unique identification (ID)

Each component shall have a unique ID in a MADES network. The identification scheme is a network governance issue.

- The "component ID" (also referred as "component code") is used to identify the component when exchanging with other components.
- A BA shall use an endpoint ID to identify the recipient when sending a document. Conversely the sender endpoint ID is provided to BA together with a received document.
- The IDs of the sender and the recipient are included in the header of each message.

When a component composes a message, it shall identify it with a UUID (Universal Unique Identifier), as defined in IETF RFC 4122 (http://www.ietf.org/rfc/rfc4122.txt).

NOTE When delivering a document, a recipient's endpoint supplies the BA with a guaranteed (i.e. authenticated) sender's identity: the component ID of the sender's endpoint. However the sender's identity is also often included within the document itself, and it is up to the BA that analyses the document to check that both identities match.

5.3 Business-type of a business-message

A MADES network may support multiple and concurrent business processes.

A party "P" having an endpoint connected to the network can operate several BAs, implementing functions to support internal activities and exchanges with others parties in accordance to the roles he plays in the business processes.

The BAs request the endpoint to send documents to other parties. The endpoint supports concurrent requests from the BAs.

Other parties, while fulfilling their own roles in these business processes, may also send some documents to party "P". For dispatching correctly the received documents between BAs, each BA may indicate a business-type when requesting for downloading a possibly newly received document.

So the business-type is mandatory text information provided by a sender's BA, included in the header of the business-message, transported with the message to the recipient's endpoint, and used by a recipient's BA to retrieve the only documents it shall process.

Each party is free to organise how he architectures activities, functions and BAs in his own Information System, in a way transparent to the other parties. However the business-types shall be agreed between all parties as part of the overall information exchange design[2].

5.4 Delivery-status of a business-message

The delivery process of each business-message is fully tracked. Tracking means that the components taking part in the routing process notifies the sender's endpoint with events about the message. The reported events are:

---

2 Business-types can be compared to port numbers: competing applications in a machine use different port numbers so that received information can be correctly routed. Used port numbers have to be agreed between parties (e.g. 21:FTP, 22:SSH, 25:SMTP) but they are not part of IP protocol which accepts any number.

- Delivery event – notifies that the business-message has been either:
  a) transferred to a component; i.e. the component confirms it received the business-message (“Transfer confirmation” is defined in 5.8);
  b) accepted by a component; the component confirms that it received the business-message and that the message successfully passed validation. It also means that the message is ready to be transferred to the next component on the route to the recipient’s endpoint (“Acceptance” is defined in 5.9).
- Failure event – notifies that a business-message cannot be delivered because:
  a) the message was rejected, for it fails the validation;
  b) or an unrecoverable error occurred when processing the message.

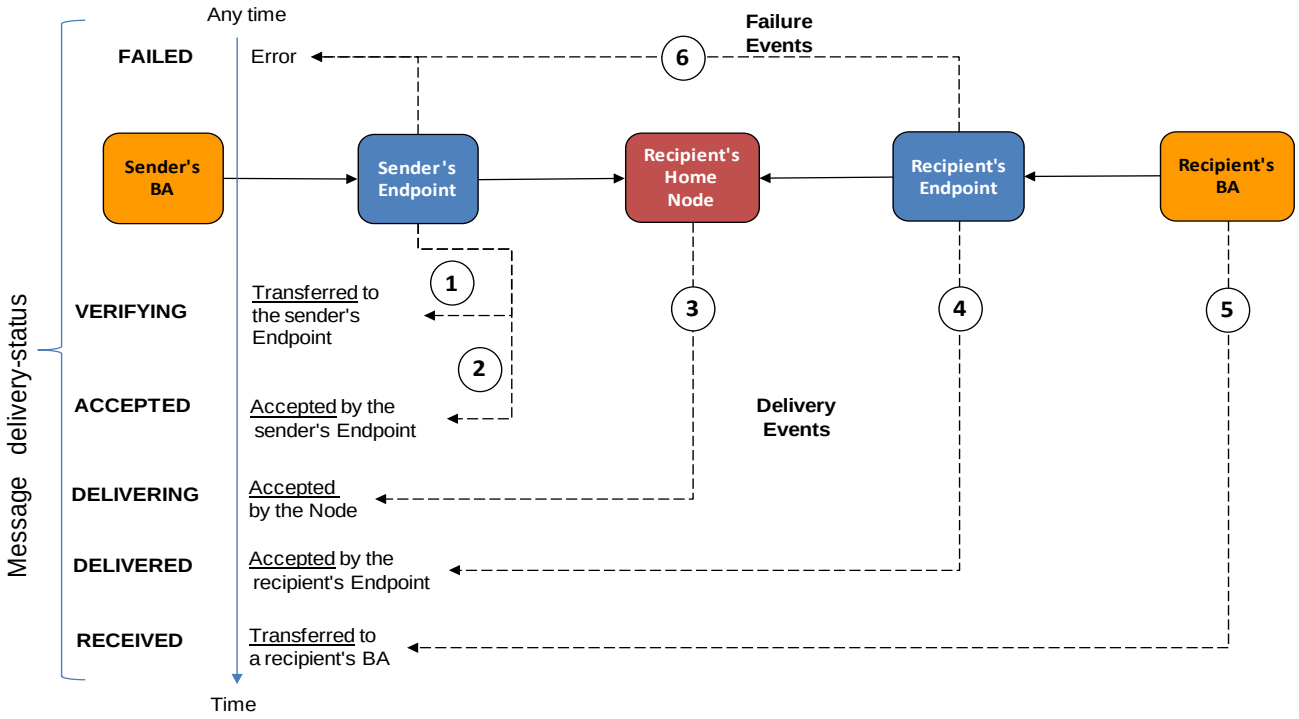

Figure 15 – Reported events during the delivery of a business-message

Figure 15 shows the possible events and the components that issue them.

The delivery-status of a business-message expresses the knowledge of the sender’s endpoint about the message delivery. The status can be requested at any moment by a sender’s BA providing the message ID returned by the sender’s endpoint when the message was sent. The possible values are provided in Table 1.

Table 1 – Message delivery status

Message delivery-status	Notified event
VERIFYING	The business-message has been transferred to the sender's endpoint. Some additional checks are in progress before the endpoint may accept it, e.g.: the endpoint is waiting for signature by an external signing device (see 5.16.3), or is waiting for the encryption certificate requested to the node directory.
ACCEPTED	The business-message has been accepted by the sender's endpoint. Conditions are met to transport the message in the network.
DELIVERING	The business-message has been accepted by the node.
DELIVERED	The business-message has been accepted by the recipient's endpoint.
RECEIVED	The business-message has been transferred to a recipient's BA.
FAILED	The message delivery has failed. So, the business-message will not be transported any further.

The acknowledgement, which notifies that a message has been accepted by the recipient's endpoint or transferred to a recipient's BA, does not and shall not mean more that the document (i.e. the content of the message) has been technically and securely delivered to the endpoint or a BA of the Information System (IS) of the recipient party. So far, the content of the document has not been analysed. The probable and further analysis may result in a "functional acknowledgment", the document being then accepted or rejected according to the business rules. Such a functional acknowledgement can even be a new document that the MADES network will be entrusted to deliver as a new business-message to the sender of the original document.

5.5 Communication between components

5.5.1 Principle

To communicate, a client component establishes a secured communication channel with a server component, and then issues requests through the channel.

The server component validates the request and replies. The client component receives back a request status and validates the reply.

5.5.2 Establishing a secured communication channel between two components

A request from a client component to a server component shall only be processed after the client has established a secured (i.e. encrypted) communication channel with the server.

The communication channel shall be secured using the HTTPS (HTTP over TLS) protocol. So each peer, either client or server, verifies that the other peer is a valid and trusted network component – see 5.16.6.

A client component shall be able to connect to a server component through a network proxy.

An endpoint administrator shall be able to configure a primary and a secondary URL to connect to the home node.

An endpoint may connect to any node for uploading business-messages and acknowledgements addressed to a recipient's endpoint registered with the connected node. The endpoint shall request by its home node directory the "routing information" for establishing connection – see 7.3.4.3.

The node URLs (primary and possibly secondary) in directory should rather contain FQDNs (Fully Qualified Domain Names) than IP addresses to ease integration with the network architecture constraints of the parties.

Concerning primary and secondary URLs: the nodes are key components and thus require high availability. Availability techniques may vary, and redundancy or switch-over mechanism may not be seamless to other components. So a node administrator may provide two URLs to access his node. Consequently, the components that connect to a node shall implement a mechanism to dynamically select the one URL which gains effective access.

5.5.3 Token authentication of the client component

Apart for the node-node synchronization, the server shall always first identify the client, i.e. know its component ID to authorise the requests.

To do so, the client shall request the server for an authentication-token providing its own component ID – see 7.3.2.

The server provides back a token which:

- is a randomly generated string (e.g. a UUID);
- has a limited duration validity returned to the client. The later has to request for a new token when expired or before the expiration time.

For every subsequent request (e.g. message transfer, directory query), the client shall always provide the server with:

- the authentication-token;
- the signed authentication-token – “signed” means that the hash of the token is encoded using the RSA algorithm – see 5.16.1;
- the ID of the authentication certificate used for signing the authentication-token.

For every received request, the server shall process the following checks:

- the authentication-token is a known and not expired token;
- the certificate used to sign is a valid and non-revoked certificate owned by the client – see 5.16.8;
- the signature of the authentication-token is correct.

NOTE Such a token-based client authentication is neither part of nor linked to the TLS authentication, and thus not constrained by specifics of software products used for the implementation of the component (e.g. web servers, applications servers).

5.5.4 Request authorisation

A node shall reject a request for downloading messages or a request on directory if the client component is not one of its registered endpoints.

A node shall reject a transfer request (download or upload) when it is already and concurrently processing the same request for the same client – see 5.8.

5.5.5 Request/Reply validation

The server shall validate data of any request and the client shall check the status and validate data of any reply.

Validation prevents for foreseeable errors to occur and shall include:

1) Check that all mandatory request/reply elements are set.

2) Checks that all set elements do not contain any illegal characters, have the expected format and size, and have values in expected list or range.

3) Check that the combination values of elements forms a valid set.

In case the request (or the reply) is a message transfer, the validation by the target component shall include any additional check to ensure that the transferred messages can be durably stored – see 5.6 (e.g. the size of the message-content does not exceed the maximal allowed size).

5.6 Storing messages in components

A component shall contain an internal message-box where it durably stores messages. Durable (or persistent) means that the message shall be recovered after a software crash or a hardware failure, when the component restarts (reboot or switch to a backup component in a redundant architecture).

The stored information about a message (either business-message or acknowledgement) shall be: the content, the header.

The endpoints shall store the compressed (if requested – see 5.14.2) but non-encrypted content of the message. The stored header shall include the message signature.

Within the message-box, a message shall be associated with additional information only used locally by the component:

- Transfer timestamp – set by the component when the message is created/stored in the message box, and used for priority management – see 5.13.
- State – see 5.7 for possible values and lifecycle.
- Priority – see 5.13.
- Receive timestamp (only used for a business-message), the time when the message was accepted by the recipient's endpoint – set when processing the acknowledgements of the message – see 5.10.4

A component administrator shall be able to configure a purge strategy for each business-type. A purge strategy indicates how the component manages a business-message that has reached the final state (see 5.7). Possible strategies should include:

- Delete only the message-content (document).
- Delete the whole message and acknowledgements.
- Never delete the message.

A component administrator shall be able to (long term) archive messages and then delete the correctly archived messages from the message-box.

5.7 Lifecycle of a message state within a component

A business-message has a local state in every component that processes it, and all these states do not have the same values at the same time. The state is not transported data; it is not part of the message header. The lifecycle of the state of a business-message within a component is shown in Figure 16.

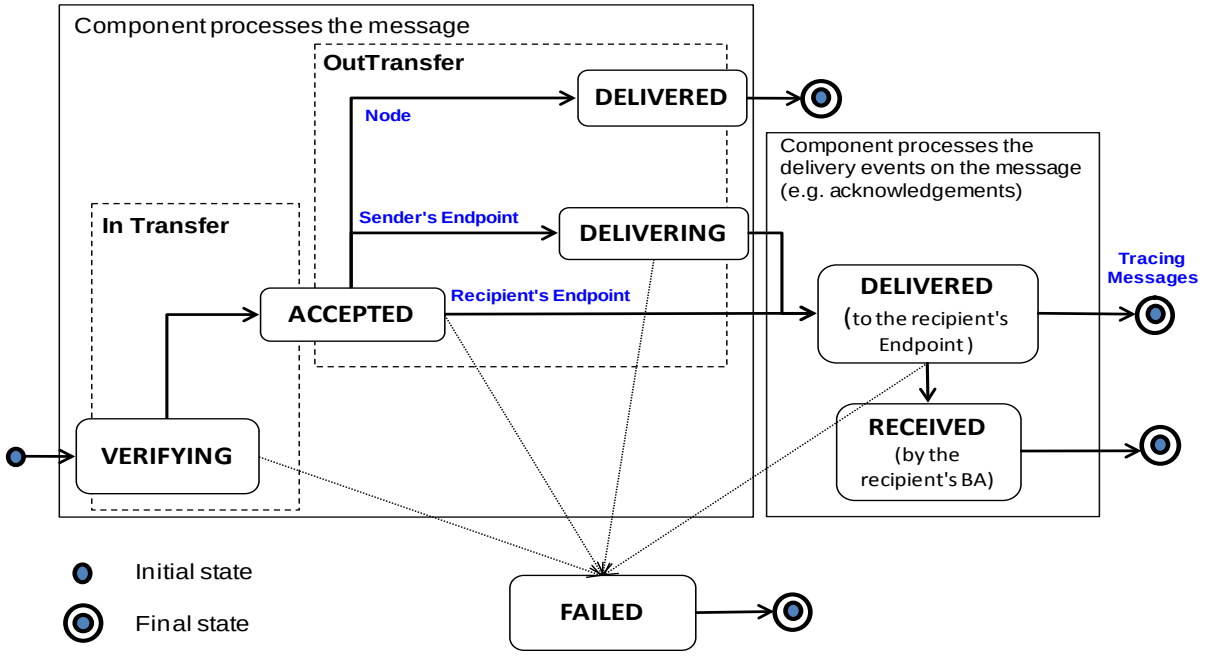

Figure 16 – Lifecycle of the local state of a business-message within a component

Possible states of a business-message are listed in Table 2.

Table 2 – Business message status

Business-message State	Description
Verifying	The successful transfer of the business-message in the component has been confirmed to the component which sent it and, before it may accept it, the message is currently passing some validation checks or pending (e.g. waiting for an external certificate to perform security operations such as signature/encryption).
Accepted	The business-message has been accepted by the component, and is pending for transfer to the next component on the message route.
Delivering	The business-message has been successfully transferred to the next component.
Delivered Received Failed	After the business-message has been transferred to the next component, the message state is set to the status of the acknowledgements coming back and which inform about the message delivery (see 5.10). A business-message in the FAILED state is not delivered. A component shall set the business-message state to FAILED when it sends a failure-acknowledgement for the message.

After a message has been successfully transferred (i.e. downloaded) from a node, the state within the node shall move to DELIVERED which is the final state (and not to DELIVERING). The reason is the following: If a node is not the home node of the sender's endpoint of a business-message, no acknowledgement will ever inform about the rest of the message delivery.

When a message is accepted by the recipient's endpoint, the state within the endpoint shall be directly set to DELIVERED, because the message has reached the destination endpoint and does not have a next component.

The delivery-status of a business-message, as defined in 5.4, is the local state of the message in the sender's endpoint.

5.8 Transferring a message between two components (Handshake)

The transfer handshake is the mechanism which ensures that no message can be lost while passing from a component to another. A component (referred as “target component”) that receives a message shall confirm to the sender component (referred as “source component”) that the message has been transferred.

A component is responsible for the message delivery from the moment it sends the transfer confirmation to the previous component until the moment it receives the transfer confirmation from the next component.

The target component confirms a message transfer to tell the source component that it took responsibility for the message, and that it should not transfer it again. It means that either:

- the message has been stored in a durable way,
- the processing of the message has generated an error that has been logged (e.g. message inconsistency).

The handshake mechanism differs whether the transfer is an upload or a download, as shown in Figure 17 and Figure 18. When downloading, the target component initiates the request and an additional request is used to confirm the transfer to the source component.

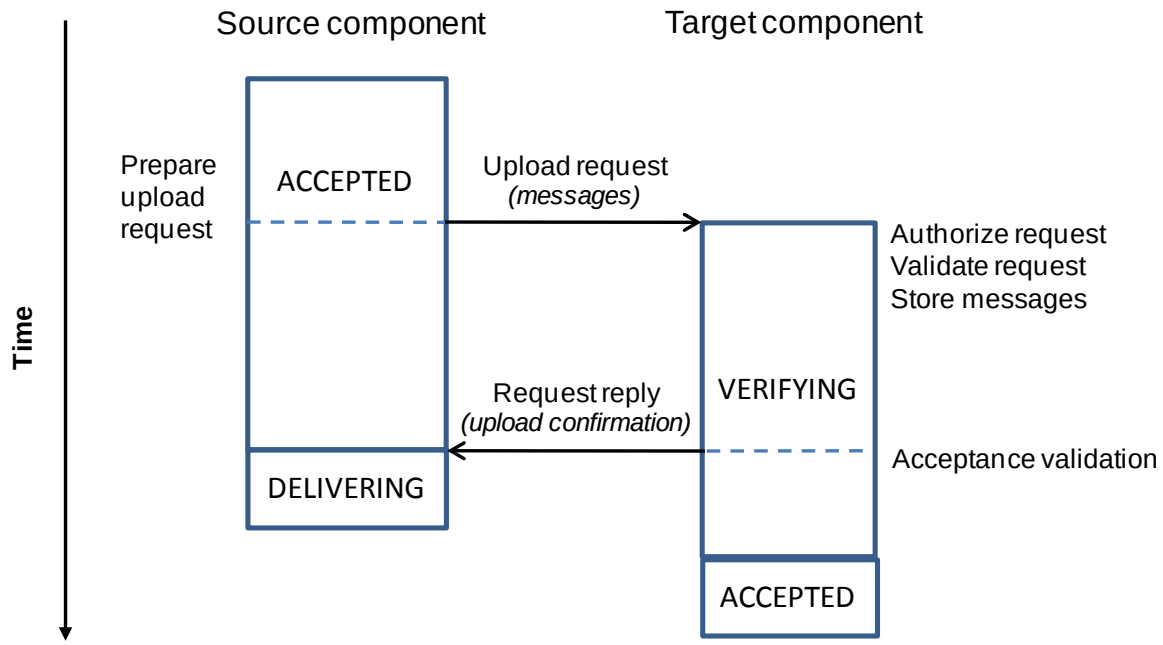

Figure 17 – Transfer handshake when uploading of a message

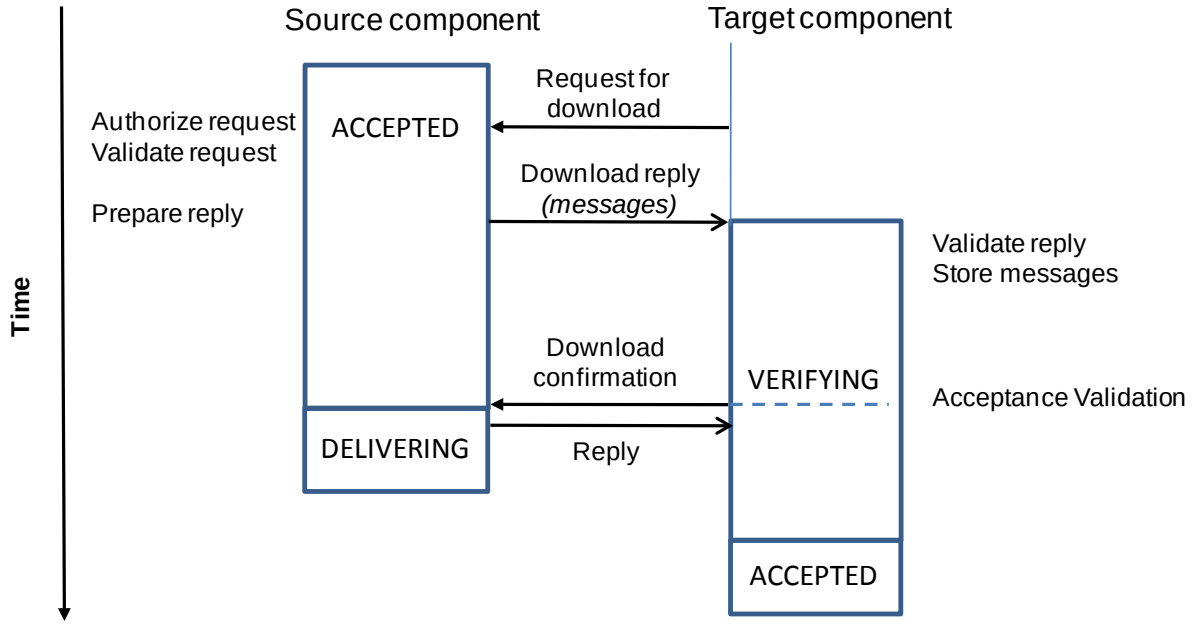

Figure 18 – Transfer handshake when downloading of a message

NOTE The use of the VERIFYING state is a component internal design issue. A target component can confirm the transfer and set the message in the VERIFYING state before asynchronously processing the acceptance checks. A target component can confirm the transfer after it processes synchronously the acceptance checks, so the message state is directly set to ACCEPTED (or FAILED) – see 5.9.

The handshake mechanism applies whether a transfer request contains one or several messages. Actually, the MADES interfaces for uploads and downloads can transfer bulk messages, mixing business-messages and acknowledgements – see 7.3.3. When multiple messages are transferred simultaneously, the confirmation shall apply to all the transferred messages. Note that the BAs can only transfer (send or receive) business-messages one by one with their endpoint.

A server component shall not authorize a transfer request from a client component while the same request from the same client is currently being processed. This is necessary to fulfil the correct delivery sequence of two messages with the same business-type – see 5.13. The bulk transfer is intended to gain performance without the use of concurrent requests.

The source component shall change the message state to the next state (generally DELIVERING) after it receives the transfer confirmation.

When the connection between the components is established or recovered after a failure, the source component shall transfer all pending messages in the ACCEPTED state. Note that it may happen that some of those messages have already been transferred, that the target component already sent the confirmation, but that the source component did not receive it or failed while processing it. So the target component may receive an already existing message (recognized with the message ID). In this situation, it shall then just confirm the transfer and log this duplicate transfer event.

5.9 Accepting a message

A component shall accept a transferred message after it passed the validation checks described in Table 3.

Table 3 – Accepting a message – Validation checks

Component	Validation checks
Sender's endpoint	The transferred message can only be a business-message: • Existence of the recipient's endpoint. • Availability of the encryption certificate of the recipient's endpoint, i.e. successfully retrieved from directory cache or from home node directory. • Successful signature of the message. NOTE The business-message shall be compressed (if requested) while received by the endpoint, and it shall be encrypted when uploaded to the node.
Node	The transferred message can be a business-message or an acknowledgement: • The recipient's endpoint has registered with the node. • The sender's endpoint exists in the directory and owns the certificate used to sign the message. • The certificates used for signing and encryption (if encrypted) exists and are not revoked (see 5.16.8).
Recipient's endpoint	The transferred message can be a business-message or an acknowledgement: • Successful decryption of the content, when encrypted. • Successful verification of the signature, when signed. • When the message is an acknowledgement notifying the event n°4 (see Figure 19), successful match between the acknowledgement content and the original message fingerprint (hash). NOTE A compressed business-message shall be uncompressed when transferring to a recipient's BA.

When a business-message is accepted, the following operations shall be processed as a transaction[3]:

- The message is updated (e.g. decrypted content; change of the local state according to the lifecycle).
- The component notifies the related delivery event – see 5.10.2.

When a business-message is rejected, the component shall notify a failure-event and update the message as a transaction.

When an acknowledgement is rejected, the component shall log the error and set the acknowledgement state to FAILED; this stops the delivery.

5.10 Event management

5.10.1 Acknowledgements

A component can notify an event which occurs when delivering a business-message, by sending an acknowledgment to the sender's endpoint of the message. The business-message on which the event occurs is referred as the original message, and its ID shall be included in the acknowledgement header.

An acknowledgement shall be routed and delivered using the same transfer (upload and download) mechanism as the business-messages, without being acknowledged itself.

An acknowledgement shall have the same business-type as the original message.

---

3 In the whole specification a "transaction" means an operation that shall succeed or fail as a complete unit and cannot remain in an intermediate state.

The content of an acknowledgement shall never be compressed or encrypted.

5.10.2 Notifying events

“Notifying an event” on a message means either:

- Sending an acknowledgement containing event information to deliver to the sender’s endpoint of the message.
- Except when the event is notified by the sender’s endpoint of the message itself, then the event information is just locally stored.

The event issuer shall update the message according the event (e.g. the message state).

Previous operations shall be realized as a transaction, and the issuer shall log the event.

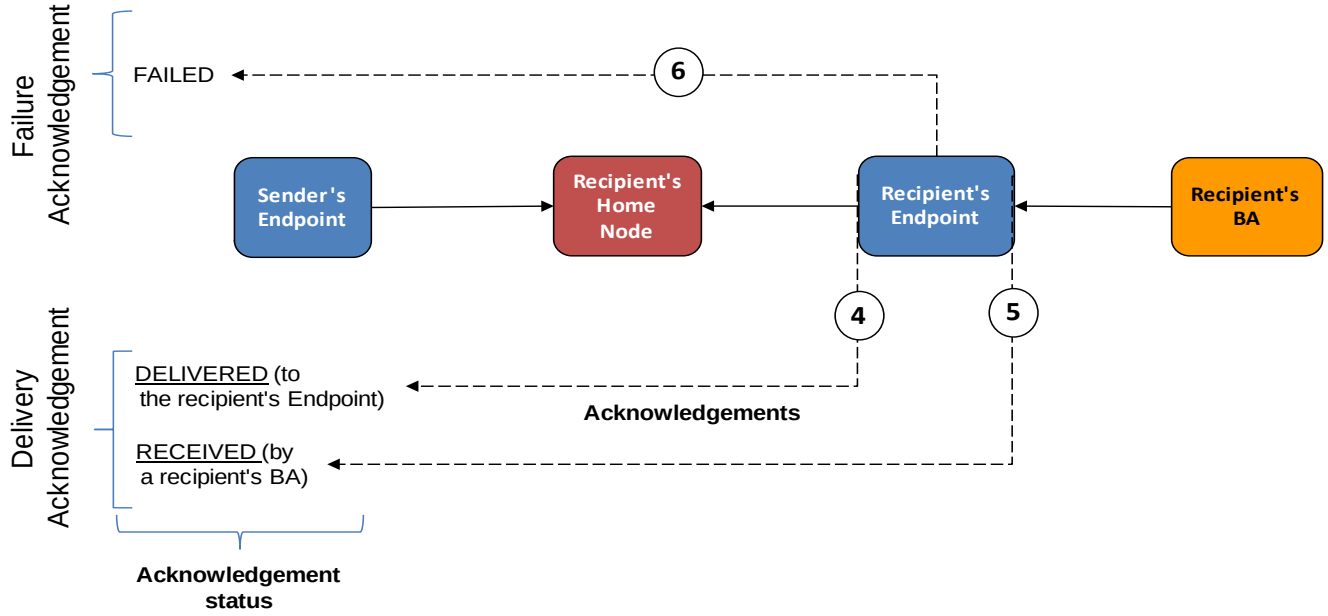

Figure 19 – Acknowledgements along the route of the business-message

Figure 19 shows the issuers and the events characteristics notified by acknowledgements. Table 4 provides the event charecteristics where the events are numbered as in Figure 19.

Table 4 – Characteristics of notified events

Event	Event characteristics
1	Status: VERIFYING Issuer: Sender's endpoint Acknowledger: None (The event is internal to the Sender's endpoint and does not generate an acknowledgement)
2	Status: ACCEPTED Issuer: Sender's endpoint Acknowledger: None (The event is internal to the Sender's endpoint and does not generate an acknowledgement)
3	Status: TRANSPORTED Issuer: Recipient's node Acknowledger: None (Although the event is coming from the node, it is notified by the sender's endpoint and does not generate an acknowledgement) Comment: A node never sends an acknowledgement because it could not be delivered if it is not the home node of the sender's endpoint of the original message. The reason is that the sender's endpoint will never connect for downloading messages, including acknowledgements. Thus the node delegates to the sender's endpoint the issuance of the acknowledgement by notifying in the upload response whether it accepts or rejects the business-message.
4	Status: DELIVERED Issuer: Recipient's endpoint Acknowledger: Recipient's endpoint ✓ Content: the non-encoded message fingerprint (hash) of the original message – see 5.14.3. ✓ Internal type: DELIVERY_ACKNOWLEDGEMENT ✓ Signed: Yes ✓ Original message state: DELIVERED Comment: State and status are set to DELIVERED because the acknowledger is the recipient's endpoint of the original message.
5	Status: RECEIVED Issuer: a recipient's BA Acknowledger: Recipient's endpoint ✓ Content: irrelevant but as least one character. ✓ Internal type: RECEIVE_ACKNOWLEDGEMENT ✓ Signed: No ✓ Original message state: RECEIVED Comment: A recipient's BA (not a MADES component) delegates to the recipient's endpoint the issuance of the acknowledgement notifying that the successful transfer of the business-message.
6	Status: FAILED Issuer: any component. Acknowledger: if not notified by the sender's endpoint ✓ Content: an English readable description of the encountered error or the reason why the message was not accepted. ✓ Internal type: FAILURE_ACKNOWLEDGEMENT ✓ Signed: No. ✓ Original message state: FAILED Comment: The event is referred as the "failure-event". In case a failure-event occurred in the sender's endpoint it processes it internally and does not send an acknowledgement.

The meaning of the characteristics is provided in Table 5.

Table 5 – Event characteristics description

Characteristic	Description
Status	The value to set to the "*state*" element of the "*trace*" item (see Table 70) reporting the event to the sender's BA through the *CheckMessageStatus* service – see 7.2.2.3.
Issuer	The component that notifies (issues) the event.
Acknowledger	The component that sends the acknowledgement, possibly none or possibly different from the issuer.
Content	The content of the acknowledgement message.
Signed	Whether the acknowledgement message is signed or not.
Internal type	The value to assign to the *internalType* element of the acknowledgement – see Table 61
Original message state	The value to set to the local state of the original message in the issuer component when issuing the event.

5.10.3 Lifecycle of an acknowledgement

Table 6 provides the possible values for the local state of an acknowledgement within a component; these are a subset of the states of a business-message.

Unless signed using an external device (see 5.16.3), an acknowledgement is created in the ACCEPTED state, otherwise in the VERIFYING state.

Table 6 – Acknowledgement state description

Acknowledgement State	Description
Verifying	The acknowledgement has been created by the component or successfully transferred to it, and a signature operation is currently processed or pending (e.g. waiting for the external device to be signed, or waiting for the external certificate to verify the signature).
Accepted	The acknowledgement is pending for transfer to the next component towards the destination endpoint.
Delivered	The acknowledgement has been successfully transferred to the next component or has reached the destination, i.e. the sender's endpoint of the original message.
Failed	The component encountered an unrecoverable error when processing the acknowledgement. The acknowledgement will never be transferred to another component.

5.10.4 Processing a transferred acknowledgement

A component shall always accept a transferred acknowledgement. When processing a transferred acknowledgement:

- The component shall log the event notified by the acknowledgement.
- In case an unrecoverable or an acceptance error (see 5.9) occurs, the component shall set the acknowledgement and the original message to the FAILED state in a transactional way, and log the error. This stops the acknowledgement delivery.
- Otherwise the component shall in a transactional way:
  - update the state of the original message according to the event in conformance to Figure 15;

– when the event is n°4 (see Figure 19), set the “receive timestamp” of the original message to the time the acknowledgement was created (generated item – see Table 61).

The original message may not exist in a node for it was delivered through another node. The acknowledgement shall then be correctly processed and route.

The original message may be in the ACCEPTED state. This may happen when the message was transferred and the component did not receive the confirmation. When the connection is back, it may receive the acknowledgement before the message is transferred again.

5.11 Message expiration

5.11.1 Principle

The message expiration is a mechanism to notify the sender’s BA that a business-message has not been delivered in the due time to the recipient’s endpoint. When the time limit is exceeded, the sender’s endpoint changes the state of the message to FAILED.

The expiration time of a business-message is the time limit when the sender’s endpoint declare that the message delivery has failed, because it has not received the acknowledgement notifying that the message was accepted by the recipient’s endpoint (event n°4 in Figure 19).

5.11.2 Setting the expiration time of a message:

An endpoint administrator shall be able to configure maximum durations for the delivery of the business-messages as:

- duration values associated to the business-types;
- a non zero and positive default duration value.

The expiration time is part of the header of a message – see Table 61, expirationTime. The time count shall start when the sender’s endpoint confirms the transfer of the business-message (event n°1). The expiration time shall be set by the sender’s endpoint according to the business-type of the message. Otherwise the default duration value shall be used.

The expiration time of an acknowledgement is the expiration time of the original message.

5.11.3 Looking for the expired messages:

Each component shall cyclically look for the expired messages either business-messages or acknowledgements. A message expires when the expiration time is past and the local state is not amongst DELIVERED, RECEIVED or FAILED.

The sender’s endpoint shall notify the expiration of a business-message using an event-failure. Otherwise the component shall set the local message states to FAILED and log the expiration (date, message ID, sender, recipient, sending time, expiration time).

NOTE The default value for maximum delivery duration is a general mechanism to set to FAILED the state of the messages whose delivery cannot be processed for whatever reason, ensuring then that they will not be forever delivering (i.e. “zombie” messages).

5.12 Checking the connectivity between two endpoints (Tracing-messages)

A tracing-message is a business-message used to check end-to-end connectivity between two endpoints using the message tracking process. The message header contains a special type for a tracing-message (TRACING_MESSAGE – see Table 62).

A BA can request to process a connectivity test with any endpoint. The sender's endpoint shall then compose and send a tracing-message to deliver to the required destination endpoint.

To check that the tracing-message reached the recipient's endpoint, the sender's BA can check its delivery status, as for any business-message.

The business-type and the content of a tracing-message are irrelevant but shall have at least one character. As any business-message, a tracing-message is signed and the content is encrypted. So the tracing-message delivery success includes the checks of the certificates' set-up and processing.

The header of the acknowledgements whose original messages are tracing-messages also have a special type (TRACING_ACKNOWLEDGEMENT – see Table 62).

Because no recipient's BA will ever request for the tracing-message, the final state of a tracing-message is DELIVERED in all components – see 5.7.

5.13 Ordering the messages (Priority)

A component administrator shall be able to configure priority values according to the business-types, and to configure a default priority value for unknown business-types.

A business-message shall have the priority configured for the business-type if defined; otherwise the default priority.

The component shall process pending messages and elaborate the transferred list of messages using the following order:

1) A message with higher priority is processed first.
2) If two messages have the same priority, the one that was first transferred by the component is processed first – see "Transfer timestamp" in 5.6.

The message priority is local to a component. It may differ between components and is not transported information.

Assume that two messages (M1 and M2) of the same business-type are sent in this order by BA1 to BA2. If BA2 receives both messages, M1 shall be received first. Whatever priority is configured for the business-type by each component, the delivery order shall remain unchanged.

An acknowledgement has the same priority as the original message, because it has the same business-type.

The priority of the tracing-messages may be configurable; otherwise they have the default priority.

5.14 Endpoint

5.14.1 Endpoint functions

An endpoint provides interfaces for BAs to send and receive messages in a secure way. An endpoint shall provide the following functions:

- Communication:
  a) Connect to a node using HTTPS.
  b) Validate the send-requests from the BAs.

c) Validate the receive-requests from the BAs and provide the received documents.

- Pre-processing the to-send business-messages:
  a) Compose the business-messages (e.g. create the message ID, set the expiration time, and compress the content).
  b) Check the existence of the recipients and get their encryption certificates by the home node directory.
  c) Generate the message signature and encrypt the message-content.
- Post-processing the received business-messages:
  a) Get the signing certificates by the home node directory.
  b) Decrypt the message-content, verify the message signature, and uncompress the message-content.
- Notifying the events on the message delivery:
  a) Send and process the acknowledgements.
  b) Verify the signature and the content of the acknowledgements.
- Processing the messages:
  a) Upload the encrypted business-messages to recipient's node.
  b) Download the encrypted business-messages from the home node.
  c) Store the messages in the local message-box.
  d) Process the validation checks.
  e) Look cyclically for the expired messages.
  f) Update the messages' states according to delivery progress.
  g) Manage the queues with messages pending for uploading or downloading.
  h) Process the messages according to local priority rules.
- Processing the tracing-messages:
  a) Validate the connectivity test requested by the BAs.
  b) Compose the message.
  c) Process the tracing-messages downloaded from the home node.
- Requesting the home node for directory information:
  a) Retrieve other endpoints signing and encryption certificates.
  b) Durably store the used signing certificates of the other endpoints.
- Replying to the messages status requests from BAs.
- Administration:
  a) Synchronize the endpoint time with a reliable source (recommendation is to use a standard OS mechanism such as NTP, the Network Time Protocol).
  b) Install endpoint and CAs certificates (initial and renewed).
  c) Archive and purge the logs.
  d) Archive and purge the messages.

5.14.2 Compression

The sender's endpoint shall compress the content of each business-message whose business-type is configured to do so.

The endpoint administrator shall be able to configure the business-types of the business-messages that shall be compressed.

The tracing-messages and the acknowledgements shall not be compressed.

Compression shall be done using the ZIP algorithm.

Compression means that the message-content is encoded and that the metadata[4] (see Table 7) shall be added to the message header – see Table 61.

Table 7 – Compression – metadata attributes

Metadata Attribute Name	Metadata Type	Description
Compression	BOOLEAN	Value:= true (i.e. the message was compressed)

The header of a non-compressed message may also contain the metadata with the attribute value set to “false”.

5.14.3 Signing

The signing principles are presented in 5.16.1.

Only the endpoints sign the messages. A sender’s endpoint shall sign every business-message. The recipient’s endpoint shall sign the acknowledgement notifying event No. 4 (see Figure 19).

An endpoint shall verify the signature of every signed message that it receives. A message is signed if it contains the signature metadata (see Table 8).

Signature algorithm shall be RSA-SHA (IETF RFC 3110 - http://www.ietf.org/rfc/rfc3110.txt). The signature format shall comply with the “XML Signature Syntax and Processing standard” (http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/).

An endpoint shall encode the message hash using the private key of the signing certificate.

The manifest used to generate the message hash is:

> Compress (*content*) + *baMessageID* + *extension* + *generated* + *internalType* + *messageID* + *relatedMessageID* + *receiverCode* + *senderCode* + *senderDescription* + *SenderApplication* + *businessType*.

Where:

- the italic names refer to the attributes of the internal message structure as described in Table 61;
- “+” is the binary concatenation of the message attributes in UTF-8 encoding;
- Note that the content of the message may be compressed or not, according to 5.14.2, but not encrypted.

Signing means that following metadata is added to the message header – see Table 61.

4 "Metadata" refers to the part of the message header named “metadata” – (see Table 7).

Table 8 – Signing – metadata attributes

Metadata Attribute Name	Metadata type	value
Algorithm	STRING	Value:= SHA-512 (*The algorithm used to generate the message hash*).
Certificate ID	STRING	The ID of the certificate whose private key was used to generate the signature, i.e. to encode the message hash – see 5.16.2.
Signature	STRING	The message signature compliant with the "XML Signature Syntax and Processing standard". The XML signature document is embedded here as a string. *(An example of an XML signature document is provided in 7.6.4).*

When receiving a message an endpoint shall check the message was signed, i.e. if the header contains signature metadata, and then:

1. Recover the signing certificate from the cache or from the home node.
2. Verify that the certificate was valid when the message was generated (Certificate expiration date-time is a certificate attribute. The generated date-time of a message is part of the message header).
3. Verify the XML signature (i.e. the "signature" attribute of the metadata) using the public key of the signing certificate.
4. Regenerate the message hash of the received message using the "Algorithm".
5. Verify that the hashes provided by operations 3 and 4 are equal.

5.14.4 Encryption

The encryption principles are presented in 5.16.1.

The sender's endpoint shall encrypt a business-message just before it is uploaded. The recipient's endpoint shall decrypt a business-message just after it is downloaded.

Only the message-content of the business-message shall be encrypted. The acknowledgments shall never be encrypted.

The encryption and decryption processes use a combination of asymmetric and symmetric cryptography, as shown in Figure 20.

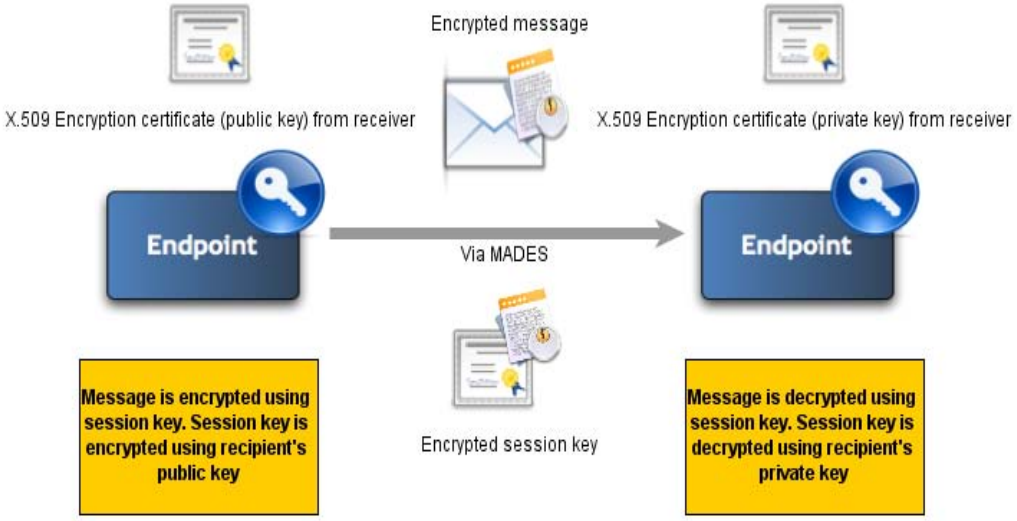

Figure 20 – Encryption process

The sender's endpoint shall use the encryption certificate of the recipient's endpoint. It retrieves it by the home node.

To encrypt the message-content, the endpoint shall first generate a random symmetric encryption key (called the session key), which is used to encode the content of the message. Then the symmetric key shall be encoded using the public key of the encryption certificate of the recipient's endpoint.

The symmetric algorithm used to encode the message shall be AES (Advanced Encryption Standard) and the key size shall be 256 bits.

Encryption means that the message-content is encoded and the metadata (see Table 9) shall be added to the message header (see Table 61).

Table 9 – Encryption – metadata attributes

Metadata Attribute Name	Metadata Type	Description
Cipher	STRING	Value:= AES-256 (*The algorithm used to encrypt the message-content with the session key*).
Certificate ID	STRING	The ID of the certificate whose public key is used to encode the session key – see 5.16.2.
Session key	BYTE_ARRAY	The value of the session key encoded using RSA algorithm and the public key of the encryption certificate.

When receiving a message the recipient's endpoint shall check if the message is encrypted, i.e. if the header contains encryption metadata, and then:

1. Verify that the certificate ID used to encrypt the message is one of the owned encryption certificates.
2. Verify that the encryption certificate was valid when the message was generated (Certificate expiration date-time is a certificate attribute. The generated date-time of a message is part of the message header).
3. Decode the symmetric session key using the corresponding private key of the certificate.

4. Decode the message-content using the decoded session key and the algorithm used for encryption.

5.15 Node

5.15.1 Node functions

A node shall provide the following functions:

- Communication:
  a) Authorize the HTTPS connections from the endpoints or from the other nodes.
  b) Connect to the other nodes using HTTPS.
- Processing the messages:
  a) Upload and download the messages to and from the endpoints.
  b) Store the messages in the local message-box.
  c) Process the validation checks.
  d) Look cyclically for the expired messages.
  e) Update the messages' states according to the delivery progress.
  f) Manage the queues of messages pending for downloading.
  g) Process the messages according to the local priority rules.
- Directory services:
  a) Provide the registered endpoints with the nodes' URLs and the endpoints' description and certificates.
  b) Request the others nodes for their reference directory data – see 5.15.2.
  c) Reply to the others nodes' synchronization requests – see 5.15.2.
  d) Manage directory data and the data version (Dversion).
- Administration:
  a) Register the endpoints.
  b) Generate the certificates for the registered endpoints
  c) Import signing and encryption certificates from externals CAs.
  d) Revoke the endpoint certificates – see 5.16.8.
  e) Import the synchronization nodes' List – see 5.15.3.
  f) Synchronize the node time with a reliable source (recommendation is to use a standard OS mechanism such as NTP, the Network Time Protocol).
  g) Archive and purge the logs.
  h) Archive and purge the messages.

5.15.2 Synchronizing directory with other nodes

A node directory is the master reference for all data regarding a sub-network composed of the node itself and the registered endpoints.

The synchronization between the nodes is carried out cyclically or on the node administrator demand. Each node requests the others nodes for their sub-network data and stores it in its own directory.

The synchronization frequency is defined by the network governance.

Example: Figure 21 shows the node A that connects to the node B, and then to the node C to obtain their directory data.

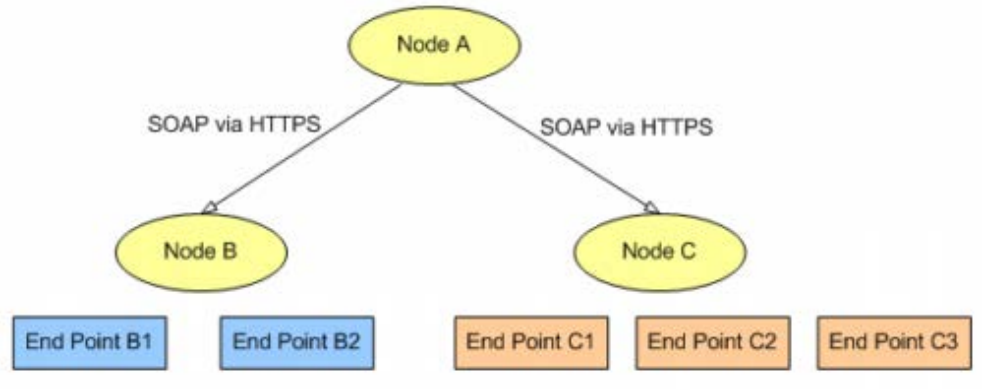

Figure 21 – A node synchronizes with two other nodes

Each node shall manage a directory version number, referred as the “Dversion” for the reference data, which increases every time they are updated.

Each node shall store reference data of the other nodes and the corresponding Dversion. The Dversion shall always be transferred together with the directory data. When requesting for data of another node, the client shall provide the Dversion of the remote data that it already possesses, so the reply can just inform that data is already up-to-date. Thus the nodes may synchronise frequently (e.g. 5 minutes).

Synchronized data shall include:

- information and certificates of all endpoints registered with the node,
- information and certificates of the node itself.

After received data has been validated (e.g. check that none of the received component ID already exist in another sub-network), the update in directory shall be a transaction.

5.15.3 Updating the synchronization nodes’ list

The network administrator is responsible to build and send to all the node administrators the list and the access information of all the nodes of the network, namely for each node:

- The node component ID.
- The node access URLs (primary, secondary).

The list is a single file, referred as the node-list file, whose format is described in 7.4.

A node administrator shall import the file to update the nodes to synchronize with.

- The node shall process the file as a transaction, i.e. any error (e.g. incorrect format, non-unique component ID, missing certificate and internal error) shall cause the rollback of the whole update process, and the directory data shall remain unchanged.
- The importation process shall ignore information about the current node which is included in the list.
- The synchronization process which may update the current node directory shall be stopped during the importation.

After importation, the node administrator can restart the synchronization process to update the directory with the reference data of the nodes.

A node shall memorize the last time it successfully retrieved the data of each node of the node-list file. Such information shall be accessible by the administrator.

NOTE A message exchanged between two endpoints having different home nodes can only be delivered correctly (including acknowledgements) after the two nodes have synchronized with each other at least once.

5.16 Certificates and directory management

5.16.1 Definitions and principles

The security of a MADES network is based on a Public Key Infrastructure (PKI). Such infrastructure binds certificates both to the network components and to the parties using the network. Indeed the components cross-check their identities before exchanging information, the sender parties want that the only intended recipients can read the documents, and each party want to authenticate the senders of the documents he received.

Certificates use asymmetric cryptography based on private and public keys. On the contrary of symmetric cryptography, encoding is done using one key and decoding using the other key (which is different, and hence the asymmetry). Where the public key can easily be deduced from the private key, the reverse operation is a very complex mathematical challenge. RSA algorithm is generally used for encoding and decoding.

Encryption:

- A document is encrypted[5] when it is encoded with a randomly generated symmetric key. The key is attached to the document in a secret way, being encoded itself with the recipient's public key.
- To decrypt the document, the recipient shall first decode the "encoded symmetric key" using its private key, and then decode the document with the symmetric key.

Signing:

- A signature is an encoded fingerprint of a list of resources. The list is referred as the signature manifest. The technical word for the fingerprint is a "hash", which is generated via a strong one-way transformation (e.g. SHA-1, SHA-512). The exact manifest of a MADES message is described in 5.14.3.
- The algorithm used to generate the hash does not require any key, so anyone having the manifest can generate the hash. Building another meaningful manifest generating the same hash is also a complex mathematical challenge. The signature is the hash encoded with the sender's private key.
- Thus anyone having the manifest, the signature and the sender's public key can verify that the manifest is the one that was manipulated by the sender when he generated the signature. The sender cannot repudiate a manifest he signed.
- Signing refers to the full process, i.e. generating the hash and encoding it.
- Verifying a signature includes: regenerating the hash from the manifest, decoding the signature, and checking that both results are equal.

A Certificate Authority (CA) is an entity that issues certificates.

A certificate:

- contains a public key, a name;
- is signed with the private key of the certificate of the issuer CA;
- has an expiration date, which is sooner than the expiration date of the certificate of the issuer CA.

---

5 Beware that "Encoding" and "Encryption" are not synonymous here. "Encoding" refers to an algorithmic operation, while "Encryption" is the process described here which ensures confidentiality. Both "Encryption" and "Signing" processes use "Encoding" operations.

When signing a certificate, the issuer CA certifies the ownership of the keys (private and public) by the party whose name is in the certificate. Other parties can verify the certificate signature using the certificate of the issuer CA. So, if parties trust a CA, they can then rely upon the signatures generated using the certificates that the CA has issued.

The certificate of a CA may itself have been issued and signed by another CA, the later delegating to the first the right to issue certificates. The certification chain of a certificate shows the delegation sequence of CAs: it is the list of the certificates of all CAs' from the issuer CA until an unsigned or self-signed certificate, referred as the root certificate.

A valid certificate is a non-expired certificate. An expired certificate shall not be used for authentication, encryption or for signing a document. However, it can still be used to decrypt old documents or verify their signatures, and thus to prove whatever may be necessary.

5.16.2 Certificates: Format and unique ID

All components (endpoints and nodes) shall use certificates to be authenticated by their communication peers (transport-layer security), to sign and to encrypt the messages (message-level security) when necessary.

The format of the certificates shall comply with the X.509 ITU-T standard, and the certificates' keys shall have a length of 2 048 bits.

The exact concatenation of the standardized attributes "issuer" and "serial number" of a certificate forms a unique ID, referred as the "certificate ID".

5.16.3 Used certificates and issuers (CAs)

5.16.3.1 Overview

Figure 22 describes the certificate authorities and the certificates used in a MADES network.

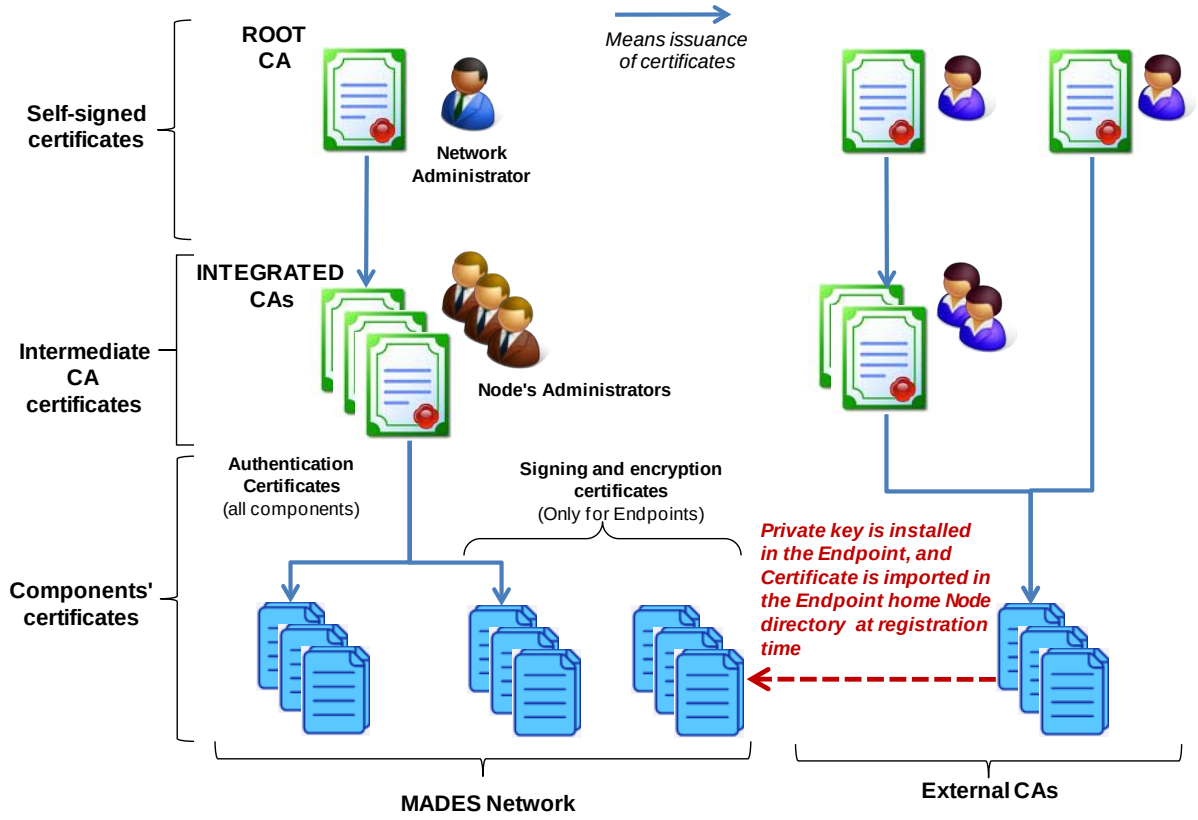

Figure 22 – Certificates and certificate authorities (CAs) for a MADES network

5.16.3.2 Transport-layer security (Authorize data exchanges)

Each component (endpoint and node) shall own an authentication certificate published in the home node directory.

The authentication certificates are issued by the network organization as follows:

- The organization owns a ROOT CA certificate.
- The organization delegates to each node administrator the right to issue the authentication certificates of the registered endpoints. Each administrator owns an INTEGRATED CA certificate issued by the ROOT CA.

Each component shall store the authentication certificate and the corresponding private key. The authentication certificate is used whether the component acts as a client or a server.

Whatever the operation using the authentication certificate, it shall fail when the certificate has expired.

5.16.3.3 Message-level security (Protect message confidentiality and authenticate message issuer)

Each endpoint shall own an encryption certificate published in the home node directory for the others endpoints to encrypt the business-messages they sent. The endpoint uses the corresponding private key to decrypt the business-messages it receives.

Each endpoint shall own a signing certificate published in the home node directory for the others endpoints to verify the signature of the messages they receive. The endpoint uses the corresponding private key to sign the messages it sends.

The signing and encryption certificates can be issued by the home node administrator using the INTEGRATED CA, or by an EXTERNAL CA trusted by the network parties and not necessary issued by one the main public trusting organizations.

The signing certificate of an endpoint can either be stored locally or inserted in a coding/decoding external device (e.g. smart cards).

The endpoint shall never encrypt or sign a message using an expired certificate. The endpoint shall not decrypt or verify a message signature using a certificate that was expired at the time the message was created (creation time is included in the message header).

Every endpoint shall durably store the signing certificates of the other endpoints in order to possess all necessary evidence.

5.16.4 Directory services

5.16.4.1 Content and updates

Each node shall contain a directory where all the network components are described. Each entry for a component in the directory shall include:

- the component ID (non-significant);
- the component display name (human readable);
- the component type (endpoint, node);
- the technical contact information for operation or administration: name of the responsible person, e-mail and phones; the latter should be non-personal (hotline, operation centre, functional/generic e-mail);
- the certificates owned by the component (one or several for each purpose including authentication, signing, and/or encryption – when applicable).

The node administrator shall be able to update the directory entry of any of the registered components. This includes registering, updating and removing components, importing or renewing certificates for components. The description of the other nodes and their registered components is imported using the node synchronization – see 5.15.2 and 5.15.3.

5.16.4.2 Queries

The endpoints shall query their home node directory to get information on a component and to retrieve certificates (e.g. to encrypt a message, to verify a message signature or to authenticate a component that signed a token or a component ID) – see 7.3.4.

5.16.5 Caching directory data

To reduce the request flow on the node directory, the endpoints shall implement a caching mechanism for directory data.

A node shall implement a TTL (Time-To-Live) mechanism, whose duration is configurable. It shall provide an expiration time for any dataset returned by a directory request: the time when the request is processed + the TTL value.

An endpoint shall not use the expired data in cache and shall then request again the home node for the data.

5.16.6 Trusting the certificates of others components

5.16.6.1 Authentication

A component shall only communicate with a peer component if the authentication certificate presented by the peer component:

- belongs to the ROOT CA certification chain;
- is successfully verified.

During the TLS authentication phase, each peer shall convey to the other the following ordered certificate chain:

1. Its own authentication certificate.
2. The INTEGRATED CA certificate certified by the ROOT CA and which certifies the authentication certificate.

A component shall trust the ROOT CA and any authentication certificate provided by the home node (e.g. used for token-authentication).

5.16.6.2 Signing and encryption

The endpoint shall trust the signing and encryption certificates provided by the home node.

5.16.7 Renewing the expired certificates

5.16.7.1 Renewing the authentication certificates

In case the authentication certificate of a component is renewed, the component will convey the certificate, possibly certified by a new INTEGRATED CA, to the peer component during the TLS authentication phase. When the INTEGRATED CA certificate is signed by the ROOT CA certificate, the communication is possible.

Additionally every component shall be configurable to communicate with components whose certificates may belong to two distinct certificate chains. So when the ROOT CA is renewed, components can communicate whether their certificate belongs to the old or to the new certification chain.

5.16.7.2 Renewing process (authentication, signing and encryption):

- The issuer CA shall renew the certificate enough time before the old one expires so that their validity periods overlap.
- The new certificate is published (imported) into the component home node.
- The new certificate is then installed into the component (including the private key), before the old certificate expires.

To do so:

- A node directory shall be able to store two certificates of any type for a component.
- Until the old encryption certificate expires, a node shall not provide the new one when replying to a directory request, for it may not have been installed into the owner component – see rules for the *GetCertificate* service; 7.3.4.2.
- An endpoint shall be able to contain simultaneously two encryption certificates (private key).

Installing a new certificate into a component shall not last more than 5 minutes to ensure business continuity.

5.16.8 Revoking a certificate

A certificate shall only be revoked for security reasons when there is a reasonable doubt that it could be misused.

Revoking a certificate is a request to the certificate issuer. As a result, the issuer usually inserts the certificate serial number in a Certificate Revocation List (CRL), which can be publicly accessed. MADES does not implement such a CRL mechanism.

Within a MADES network, revoking a certificate is a request to the administrator of the node where the endpoint has registered. The node administration tool shall provide the ability to revoke any certificate of a registered endpoint. The certificate shall then be tagged as revoked in the node directory[6]. This tag is:

- propagated to others nodes by the node synchronization mechanism;
- used to decide whether a requested certificate is delivered or not – see 7.3.4.2.

The node administrator shall be able to revoke a certificate either issued by the INTEGRATED CA or by an EXTERNAL CA.

Such revocation of a certificate issued by an EXTERNAL CA has no link with the revocation process stated by the issuer in his certificate policy. The certificate owner shall also and always and independently ask for the certificate revocation by the certificate issuer. No MADES components ever access to any CRL of an external issuer.

The consequences of a certificate revocation, resulting from the message delivery mechanism described in the previous sections, are summarized in Table 10.

Table 10 – Consequences of a certificate revocation

Revoked certificate	Consequences
Endpoint signing certificate	From the revocation moment, all business-messages and all signed acknowledgements coming from the endpoint and uploaded to the home node will be rejected. The business-messages and the signed acknowledgements coming from the endpoint and uploaded to another node will be rejected after the node has synchronized with the endpoint home node. The messages to and from the endpoint pending in a node (home or not) before the revocation tag is updated in the node, will be delivered.
Endpoint encryption certificate	From the revocation moment, all business-messages for the endpoint and uploaded to the home node will be rejected. The business-messages for the endpoint pending in the home node before the revocation moment will be delivered to the endpoint.
Endpoint authentication certificate	From the revocation moment, the home node will reject downloading messages for the endpoint. Those messages will be delivered (if not expired) after the endpoint has renewed the certificate.

NOTE The process to renew a revoked certificate is defined by the network governance.

6 This shall increase the directory Dversion – see Clause 6.

6 Managing the version of the MADES specification

6.1 Issues and principles

6.1.1 General

When the MADES specification changes from version N–1 to version N, a MADES network should then upgrade to the new version.

A “big bang” rollout on all components would be both complex to coordinate and risky regarding business continuity, and thus inacceptable.

A smooth rollout means that an upgraded endpoint can successfully exchange messages with a non-upgraded endpoint (using version N–1), and that two upgraded endpoints can successfully exchange messages using new version N.

This clause shows how such a rollout shall be done and the constraints that any new version of the specification should satisfy.

6.1.2 Rolling out a new version (Mversion and N-compliance)

The rollout of a new version shall start with nodes. An endpoint shall only upgrade after the home node did.

A node upgraded to version N can successfully process the requests from the non-upgraded endpoints if and only if it still exposes interfaces compliant with version N–1. So, as a general rule, a node upgraded to version N shall still expose the N–1 compliant interfaces. Also the upgraded nodes shall still request, being clients, the non-upgraded nodes for synchronization.

A component is referred as N-compliant when it complies with version N of the MADES specification, and when it can successfully transfer messages with components which comply with version N–1.

From version 2, every component shall be N-compliant. This means that it complies with a version and can exchange using the previous version.

Every component shall access the installed version to which it complies, referred as the Mversion (MADES version).

Notations:

- A N-service or a N-interface is a service or an interface that complies with the version N of the MADES specification.
- A N-component is a N-compliant component (e.g. N-node, N-endpoint). Note that a N-component exposes (as server) or uses (as client) N-services and N–1-services.
- A N-message is a message composed according to the version N of the MADES specification.

6.1.3 Service compatibility

A N-component server exposes both N-interface and N–1-interface. This does not mean that the N-interface is completely new (e.g. some services may not change).

It is up to the specification team to decide which and how the functions, the interfaces and the services evolve. The possible changes for a service are listed in Table 11.

Table 11 – Service compatibility – Possible changes

N°	Service changes
1	The service does not change.
2	The description of the service does not change but the way the elements are used in queries and responses does change, e.g.: • Some previously technically optional elements are now functionally required. • New values are now possible in the elements (e.g. new encryption algorithm using different metadata).
3	The description of the service changes in a compatible way, e.g.: • A new optional element is created. • A mandatory element becomes optional. • An unused optional element is removed.
4	The description of the service changes in a non compatible way → Actually, it is a new service with a new name.

In order to allow the specification team to use all these possibilities, most services include a "*serviceMversion*" element as part of the request. So the service behaviour can change without creating a new service, provided the client uses that element to tell the server which version of the specification it is working with. The server can then process the request and reply as expected.

6.1.4 Message compatibility

The description of a message or the way a message is composed may evolve from version N–1 to version N. When this happens, a N–1-component will probably fail to process a N-message.

To ensure that a sender's endpoint composes a message that a recipient's endpoint can understand, the principles are as follows:

- The node directory shall store (dynamically) the installed Mversion of the registered endpoints, which is then transferred to other nodes through the synchronization process.
- Each endpoint shall notify its installed Mversion to the home node when starting using the *SetComponentMversion* service.
- The directory services shall provide the installed Mversion of an endpoint.
- The description of a message contains a *messageMversion* element which tells the version of the specification to which the message complies.
- The transfer (upload and download) N-services shall mix N-messages and N–1-messages, e.g. the collection of messages transferred in the *UploadMessage* request can contain both N-messages and N–1 messages.
- A sender's endpoint shall compose a business-message that the recipient's endpoint can understand – see detailed rules in 6.2.4.
- A component shall compose an acknowledgment using the same Mversion as the original message.
- In case a component receives a message that it cannot process:
  a) It shall reject the message, while confirming the transfer when a node.
  b) Otherwise it shall log the error and (if possible) it may store the message in the FAILED state, and shall issue a failure-event.

6.1.5 Interface with BAs

The BAs are not concerned with the MADES specification version; so the used Mversion is not an element of the endpoint interface.

A change in a service of the endpoint interface should be backward compatible; otherwise the new specification should create a new service, and both (old and new) services should be described in the new specification. New BAs would then use the new service, and existing applications would migrate to the new service. Thus the migration timescale for the BAs can be kept independent of the network components' upgrade.

An endpoint administrator shall be able to configure an association between a business-type and a minimum required Mversion. The default value is 1. It can be used when some new features available from this version are required for the business process (e.g. new encryption algorithm).

6.2 Using the correct version for services and messages

6.2.1 Node synchronization and authentication

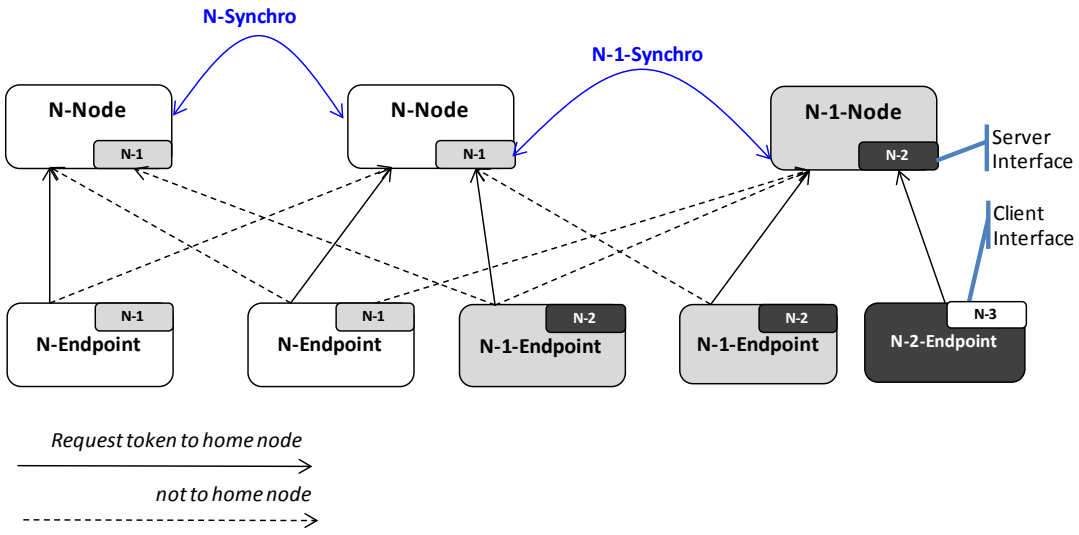

Figure 23 – Managing the specification version – node synchronization and authentication

Figure 23 shows which version of the authentication and synchronisation service is used between components. A N-component server also exposes a N–1-interface and, acting as a client can request the N–1-interface of a N–1-component server.

Node synchronization:

- A node shall request and store the Mversion of each node of the node-list file.
- The *GetNodeMversion* service (see 7.3.5.1) shall be used by a node to get the Mversion of a node of the node-list file, each time the node (re)starts and each time the node-list file is updated.
- A N_A-node shall stop synchronizing with a N_B-node when $|N_A–N_B|>1$[7].
- When requesting for a N_B-node directory data using the *GetAllDirectoryData* service (see 7.3.5.2), a N_A-node shall use the N-service, where N = Min (N_A, N_B)[8].

Requesting for an authentication-token:

- A N_E-endpoint shall request for a token to the home node using the N_E-service.

---

7 |a|: means "absolute value" (or "modulus") of a.

8 Min (a, b): means "minimum" value of a and b.

- A N_E-endpoint shall request for a token to another N_N-node using the N-service where N = Min (N_E, N_N). The Mversion of the node is available in the home directory with the node routing information.
- A N_N-node shall reject the authentication request from a N_E-endpoint when (N_E>N_N) or (N_E<N_N–1).

6.2.2 Directory services and Network acceptance

Figure 24 describes the management of different MADES version in a MADES network.

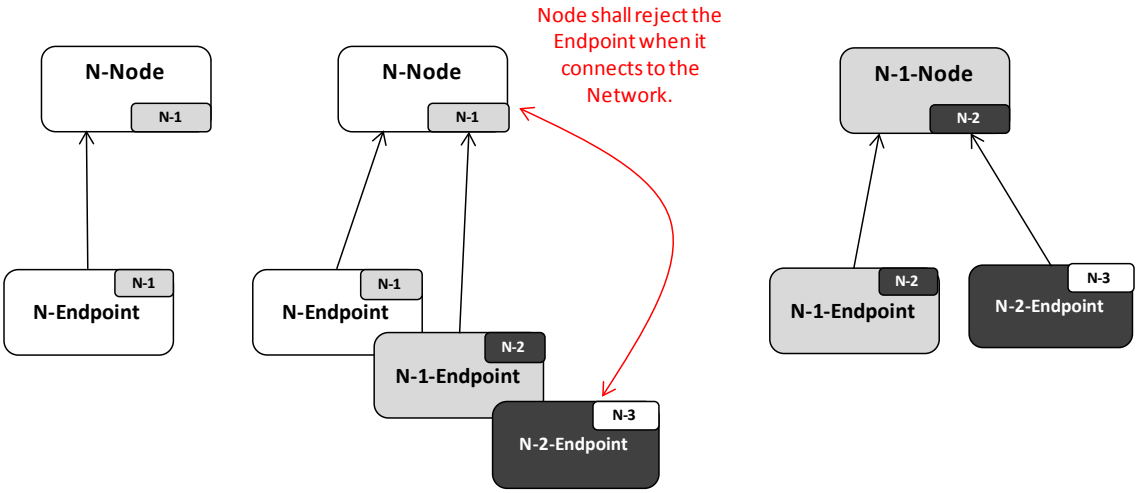

Figure 24 – Managing the specification version – Directory services

A N_E-endpoint shall always use the N_E-interface when requesting a directory service.

After an endpoint has obtained an authentication-token from the home node, it shall always request for acceptance in the network.

To do so, the component uses the *SetComponentMversion* service (see 7.3.4.1) to inform the server about its installed Mversion. The reply informs the endpoint whether it is accepted or rejected by the network.

A rejected component shall log the error and stop running.

Acceptance by the home node

- A N_N-node shall reject a N_E-endpoint when either:
  a) (N_E>N_N) or (N_E<N_N–1);
  b) the node cannot authenticate the endpoint (i.e. incorrect signed endpoint ID);
  c) the endpoint did not register with the node;
- Otherwise the node shall accept the endpoint, store the endpoint Mversion in the directory, increase the directory data version (Dversion) if N_E stored value changes, and reply providing its own Mversion (N_N).
- A node shall log that the Mversion of an endpoint has changed; or that the endpoint has been rejected.

When a node restarts, after the installed version has changed, no session information (e.g. token) from the previously connected endpoints shall remain. This ensures that all endpoints will newly request for network acceptance.

NOTE In case an N–1-endpoint is stopped, other endpoints will continue to send it N–1-messages. When it comes back to the network, being upgraded to version N, other endpoints will still continue to send it N–1-messages until their directory cache (see 5.16.5) is renewed. But the endpoint will process correctly those N–1-messages. Only the pending N–2-messages will be rejected, but anyway the endpoint cannot exchange anymore with those N–2 peers until they upgrade.

6.2.3 Messaging services

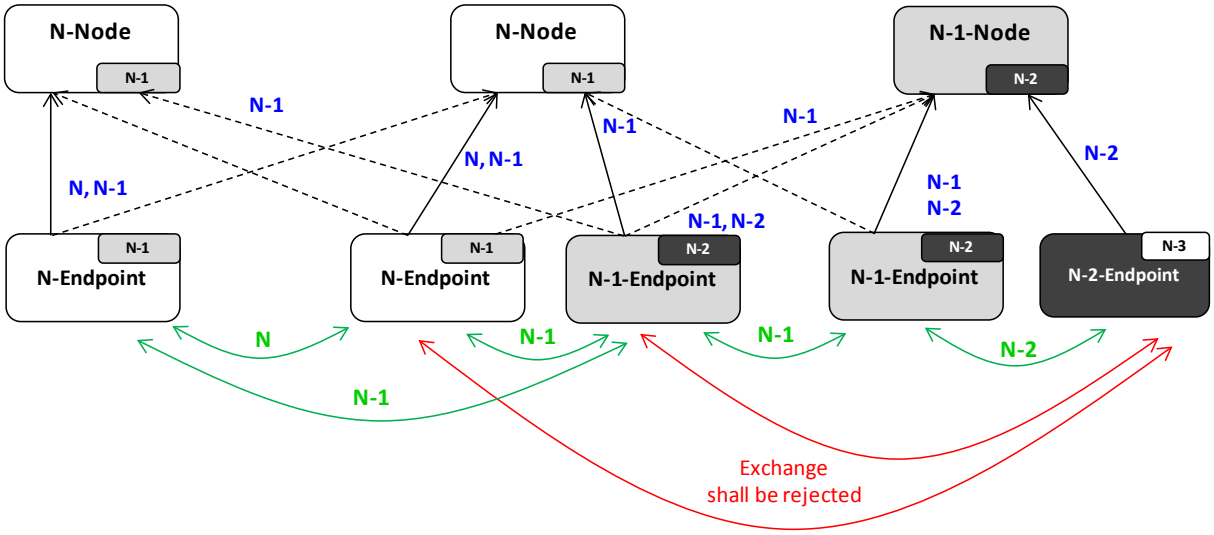

Figure 25 – Managing the specification version – Messaging services

Figure 25 shows:

- the messaging services that shall be used between components and the possible Mversion of the transferred messages (in blue);
- the endpoints that can exchange messages and the required Mversion for the exchange (in green).

Figure 25 presents a situation where two endpoints cannot exchange although they only have 1 version difference (N–2, N–1). The reason is that the N–1-endpoint has registered with a N-node. And the N-node will reject any N–2-message either a business-message or an acknowledgement.

6.2.4 Which version to use to send a message?

Figure 26 describes the way of handling different versions of MADES and Table 12 provides the meaning of the references.

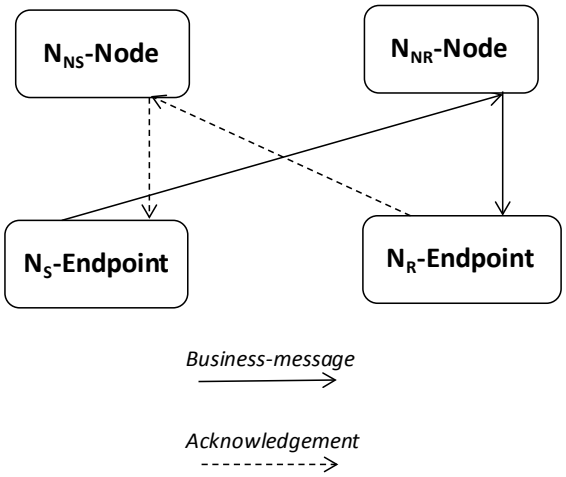

Figure 26 – Managing the specification version – Which version to use to send a message?

Table 12 – Which version to use to send a message?

Mversion	
N_S	The Mversion of the sender's endpoint.
N_{NS}	The Mversion of the home node of the sender's endpoint.
N_R	The Mversion of the recipient's endpoint.
N_{NR}	The Mversion of the home node of the recipient's endpoint.
N_B— see 6.1.5	The minimum Mversion required for the business-type

The used version should be N = Min (N_S, N_R), however the message shall be rejected if one of the conditions listed in Table 13 is verified.

Table 13 – Managing the specification version – Rejection conditions

Condition	Reason for rejection
N_R unknown	The MADES version of the recipient's endpoint is unknown.
$\|N_R - N_S\| > 1$	The sender's endpoint and the recipient's endpoint are not MADES compatible.
$\|N_{NR} - N_S\| > 1$	The sender's endpoint is not MADES compatible with the recipient's node
$\|N_{NS} - N_R\| > 1$	The recipient's endpoint is not MADES compatible with the sender's node.
$N_B > N_S$	The sender's endpoint is not MADES compatible with the minimal version required for the business-type
$N_B > N_R$	The recipient's endpoint is not MADES compatible with the minimal version required for the business-type.

7 Interfaces and services

7.1 Overview

7.1.1 General

This chapter describes all services for components to exchange each other or with the BAs. The description provides all elements in a request and the corresponding reply independently of an implementation language.

7.1.2 Error Codes

In case a service encounters an unrecoverable error, it returns information on the error. When not described the set of the returned elements is listed in Table 14 and the errorCode values are listed in Table 15.

Table 14 – Interfaces and services – Generic error

Element name	Description	Element type
errorCode	A code representing the type of error.	string
errorID	Unique identification of the error.	string
errorMessage	An English readable text describing the error.	string
errorDetails	(optional) Additional English readable details about the error context.	string

Table 15 – Interfaces and services – String value for errorCode

String value for errorCode	Description
INVALID_PARAMETERS	The provided parameters (i.e. request elements) are incomplete, are not in the expected format or do not have the expected syntax.
AUTHENTICATION_ERROR	The peer component cannot be authenticated
VALIDATION_ERROR	The message is not valid (content size exceeded, unknown sender/recipient, signature is not valid etc.).
INTERNAL_ERROR	Internal application error. The error was not caused by the content of request but by the application itself (*Null Pointer Exception* in code, full database etc.)
CONCURRENT_ERROR	The server component is already processing a concurrent request from the same client.

7.1.3 Types for Time

All date and time shall be expressed in UTC (Coordinated Universal Time). The used time types are:

- “timestamp” technically means “xsd:long”, and the value is the number of milliseconds since 'midnight 1.1.1970 UTC'.
- “dateTime” technically means “xsd:dateTime” and the value is according to the XSD specification (http://www.w3.org/TR/xmlschema-2/#dateTime).

7.2 Endpoint interface

7.2.1 Overview

The endpoint interface provides the business applications (BAs) with the access to the MADES communication network.

MADES specifies this interface using Web services – The BA calls the web services exposed by the endpoint.

There are 5 available services:

- *SendMessage* – used to upload a message into the endpoint in order to send it to another endpoint.
- *ReceiveMessage* – used to download a message from the endpoint.
- *CheckMessageStatus* – used to check the current delivery status of a message.
- *ConnectivityTest* – used to check if another endpoint can be reached.
- *ConfirmReceiveMessage* – used to notify the endpoint that a received message has been technically accepted by a BA.

The BAs can access the network using files. This interface is called FSSF (File System Shared Folders) and is described in 7.2.3.

7.2.2 Services

7.2.2.1 SendMessage service

The *SendMessage* service is used by a BA to upload a message into the endpoint in order to send it to another endpoint.

The service request elements are provided in Table 16.

Table 16 – SendMessage – Service request elements

Element name	Description	Element type	Required
message	Sending context, content and requested destination of the message.	SentMessage (see Table 76)	True
conversationID	Unique identifier associated with the request.	string	False

Concerning *conversationID*: There are situations where the sender's BA may not receive back or may fail to durably store the returned message ID, for example in case of failure of the endpoint, of the network or of the BA itself. So the BA does not know if the message was or was not correctly transferred to the sender's endpoint. There are two subsequent issues:

- If the message was actually correctly transferred and stored in the endpoint, the BA does not know the message ID needed for further processing, such as checking the delivery status of the message.
- Considering that losing a message is a non acceptable risk, the BA will send the message again when the connection with the endpoint is restored. The drawback is that the same message may then be sent twice with two different IDs.

So, resending the message using the same *conversationID* value solves both issues. Indeed, when an endpoint is requested to send a message with a *conversationID* value that has already been used for an existing stored and sent message, it shall not send the message again but return the caller BA with the ID of the already existing message. The recommendation is that *conversationID:= senderApplication + baMessageID.*

The service response elements are provided in Table 17.

Table 17 – SendMessage – Service response elements

Element name	Description	Element type
messageID	The UUID (Universal Unique ID) of the message composed and stored by the endpoint – see 5.2.	string

Additional[9] error elements for the service are listed in Table 18.

Table 18 – SendMessage – Additional error elements

Element name	Description	Element type
receiverCode	The component ID of the requested recipient's endpoint for the message.	string

7.2.2.2 ReceiveMessage Service

The *ReceiveMessage* service is used by a BA to download a message from the endpoint.

The service request elements are provided in Table 19.

Table 19 – ReceiveMessage – Service request elements

Element name	Description	Element type	Required
businessType	The business-type of the requested message – see 5.3. Pattern: [A-Za-z0-9]+ [10]	string	True
downloadMessage	The service returns, if any, the first received and pending message having the requested business-type. "First" means according to the priority defined in 5.13. The content (or document) of the message is or is not returned according to the value of the element: true:= returned; false:= not returned.	boolean	True

The service response elements are provided in Table 20.

Table 20 – ReceiveMessage – Service response elements

Element name	Description	Element type
receivedMessage	Sending context and possibly content of a message.	ReceivedMessage (see Table 74)
remainingMessagesCount	The number of remaining messages received by the endpoint, matching the requested business-type and waiting for delivery. In case the service returns the content of a message, the message is not included in the count of the remaining messages.	integer

[9] In addition to the elements described in 7.1.2.

[10] Pattern is the « regular expression » that the element value shall match.

Additional error elements for the service are provided in Table 21.

Table 21 – ReceiveMessage – Additional error elements

Element name	Description	Element type
businessType	The business-type that was requested.	string

Until the recipient's BA confirms to the recipient's endpoint that the message is correctly transferred using the *ConfirmReceiveMessage* service, the endpoint shall consider that the message has not been transferred, but is still pending and shall be transferred again next time a BA requests for the business-type. This ensures that no message may be lost. As a consequence the BAs shall be aware that, in some failure or recovery situations, they may possibly receive an already delivered message (i.e. having a known message ID).

7.2.2.3 CheckMessageStatus Service

The *CheckMessageStatus* service is used to check the current delivery status of a message.

The service request elements are provided in Table 22.

Table 22 – CheckMessageStatus – Service request elements

Element name	Description	Element type	Required
messageID	The UUID (Universal Unique ID) of the message whose status is requested – see 5.2.	string	True

The service response elements are provided in Table 23.

Table 23 – CheckMessageStatus – Service response elements

Element name	Description	Element type
messageStatus	All Information about the message delivery.	MessageStatus (see 7.5)

Additional error elements for the service are provided in Table 24.

Table 24 – CheckMessageStatus – Additional error elements

Element name	Description	Element type
messageID	The requested message ID.	string

7.2.2.4 ConnectivityTest Service

The *ConnectivityTest* service can be used to check if another endpoint can be reached. The service just sends a tracing message whose delivery-status can further be requested using the *CheckMessageStatus* service. The connectivity is successful, i.e. the tracing-message has reached the recipient's endpoint, when the status is DELIVERED.

The service request elements are provided in Table 25.

Table 25 – ConnectivityTest – Service request elements

Element name	Description	Element type	Required
receiverCode	The component ID of the recipient's endpoint whose connectivity is checked. Pattern: [A-Za-z0-9-@]+	string	True

The service response elements are provided in Table 26.

Table 26 – ConnectivityTest – Service response elements

Element name	Description	Element type
messageID	The message ID of the tracing-message.	string

Additional error elements for the service are provided in Table 27.

Table 27 – ConnectivityTest – Additional error elements

Element name	Description	Element type
receiverCode	The component ID of the recipient's endpoint whose connectivity check was requested.	string

7.2.2.5 ConfirmReceiveMessage service

The *ConfirmReceiveMessage* service is used by a recipient's BA to confirm the download transfer of a message from the recipient's endpoint.

A BA cannot reject a message; the business functional acceptance (i.e. compliance with business rules) is another issue. If a message is not confirmed back, for example in case of failure, the endpoint will provide it again at the next *ReceiveMessage* call.

In case a BA encounters an unrecoverable error when processing a transferred message, and when the error comes from the message itself (e.g. inconsistent elements) and not from the application (e.g. file system full), the BA should confirm the transfer, log the error and possibly alert, otherwise the message will indefinitely be retransferred by the endpoint until it is confirmed.

The service request elements are provided in Table 28.

Table 28 – ConfirmReceiveMessage – Service request elements

Element name	Description	Element type	Required
messageID	The UUID (Universal Unique ID) of the message whose transfer to the BA is being confirmed – see 5.2.	string	True

The service response elements are provided in Table 29.

Table 29 – ConfirmReceiveMessage – Service response elements

Element name	Description	Element type
messageID	The UUID (Universal Unique ID) of the message whose transfer has be confirmed.	string

Additional error elements for the service are provided in Table 30.

Table 30 – ConfirmReceiveMessage – Additional error elements

Element name	Description	Element type
messageID	The requested message ID.	string

7.2.3 File System Shared Folders (FSSF)

7.2.3.1 Overview

The FSSF interface is the way for a BA to exchange documents as files with the endpoint. The file system where the files are written is accessed by the endpoint as local file system. The principles are the followings:

- All the sender's BAs write in a common and unique OUT-folder the documents that the endpoint shall send.
- The recipient's BAs read in an IN-folder the documents that the endpoint has received.
- Additional information that is necessary for the message delivery is included in the filenames. Such information is the request/reply elements of the *SendMessage* and *ReceiveMessage* services.
- The organisation of the directories is local to each endpoint and is configurable.

When implemented, a file interface with the endpoint shall comply with the FSSF interface as described in the current section.

NOTE There are differences between interfacing the endpoint using FSSF and using the webservice interface.

- *CheckMessageStatus* and *ConnectivityTest* services are not supported.
- *ConfirmReceiveMessage* is implicit; i.e. a message is moved to the RECEIVED state in the recipient's endpoint when the content has been successfully written into a file in the IN-folder.
- Actually FSSF, i.e. the processing of sending and receiving documents using files, may be considered —and also probably built— as a business application (BA) embedded with the endpoint.

7.2.3.2 Used files and file naming convention

There are 4 types of files used by the FSSF interface. Each file type is written into a separate folder.

The filenames are built from several parts joined by underscores ("_").

- Each part is limited to alphanumeric or hyphen characters. Accented characters, white spaces, and special characters shall not be used.
- Joining underscores shall be present even when optional part is missing or empty.

Table 31 provides respectively the description and filename format and Table 32 the filename description.

Table 31 – FSSF – Description and filename format

Type	Folder / Writer	Description and Filename format
Files to be sent	OUT / BAs	The folder contains the files written by BAs to be sent by the endpoint. The sender's endpoint removes from the folder the files that it has processed correctly (i.e. accepted files are deleted) or not (i.e. rejected files are moved to the OUT_ERROR folder). Filenames: <SenderBA>_<Recipient>_<BusType>_<BAmessageID>.<Ext>
Failed files	OUT_ERROR / Sender's endpoint	The folder contains the files that the sender's endpoint did not process correctly. They have been moved from the OUT-folder to the OUT-ERROR-folder without changing their names. It's up to the endpoint administrator to analyse and clean up the folder. The filenames can match or not the "files to send" filename format. Note that not matching the filename format is a reason for the file not to be processed correctly.
Received files	IN / Recipient's endpoint	Each file in the folder contains the content of a message that has been received by the recipient's endpoint. The filename is built from the header information of the received message. The files should be removed from the folder when processed, correctly or not, by the recipient's BAs. Filenames: <SenderBA>_<Sender>_<BusType>_<BAmessageID>_<MessageID>.<Ext>
Log files	OUT_LOG / Sender's endpoint	The folder contains one log file for each message accepted by the endpoint. The file contains English readable text. Each line reports an event about the message delivery, and is the concatenation of the *MessageTraceItem* structure, as provided in the *CheckMessageStatus* service response. The file is appended with a new line each time a new event for the message is notified to the sender's endpoint. It's up to the endpoint administrator to clean up the OUT_LOG folder. The filename is the exact name of the sent file with an added ".log" extension: <SenderBA>_<Recipient>_<BusType>_<BAmessageID>.<Ext>.log

Table 32 – FSSF – Filename description

Filename parts	Type	Description
<BAmessageID>	Optional	An identifier of the document provided by the sender's BA. Information is transported "as is" to the recipient's BA. – Pattern: [A-Za-z0-9-]*
<BusType>	Mandatory	The business type for the message (see 5.3). – Pattern: [A-Za-z0-9-]*
<Ext>	Optional	The file extension – Pattern: [A-Za-z0-9-]*
<MessageID>	Mandatory	The UUID (Universal Unique ID) of the message composed by the sender's endpoint (see 5.2) – Pattern: [A-Za-z0-9-]+
<Sender>	Mandatory	The component code of the sender's endpoint. – Pattern: [A-Za-z0-9-@]+
<SenderBA>	Optional	The identifier of the sender's BA. – Pattern: [A-Za-z0-9-]*
<Receiver>	Mandatory	The component code of the recipient's endpoint. – Pattern: [A-Za-z0-9-@]+

Additional rules:

- The to-be-sent filenames without extension shall not end with the dot character (".").
- The sender's endpoint shall ignore files in the OUT-folder with extension "TMP" (or "tmp").
- The sender's endpoint shall fail to send the files that matches one of the following conditions:
  1) Filename does not match the "files to-be-sent" Filename format.

2) Filename is longer than 200 characters.
3) File is empty.

7.2.3.3 Concurrent access to files

7.2.3.3.1 General

As the BAs and the endpoint concurrently access to the files, special attention is required to avoid access conflicts and data losses.

7.2.3.3.2 Access conflicts

To avoid access conflicts between the writer and a reader of the file:

- The file-reader shall ignore files whose extension is “TMP” (or “tmp”).
- The file-writer shall write the data first in a temporary file whose extension is “TMP” (or “tmp”), and then rename it changing the extension (note: “rename” is an atomic operation on every file system).

7.2.3.3.3 Data losses

Data may be lost if a file is overridden by another file having the same filename. To avoid this, each file should have a unique filename:

- OUT – OUT_ERROR – OUT_LOG: It is highly recommended that the BAs uses <SenderBA> and <BAmessageID> to ensure they cannot use the same filename.
- IN: the use of <MessageID> in the filename ensures that the content of two different messages will always be written in 2 different files.

7.2.3.4 Configuring FSSF

The administrator shall be able to configure the endpoint with following information:

- OUT folder name;
- OUT_ERROR folder name;
- OUT_LOG folder name;
- A list of business-types, and for each one an associated folder name and a default extension.

The recipient’s endpoint shall write in files the content of the business-messages whose business-type is in the configured list:

- The file shall be written in the IN folder which is associated with the message’s business-type.
- The file shall have the extension provided in the “extension” attribute of the message header – see Table 61. When none, the extension shall be the default extension for the message’s business-type.

7.3 Node interface

7.3.1 Overview

The node interface (see Figure 27) provides the endpoint access to the node. MADES specifies the interface exposed by the node using Web services – over SOAP/HTTPS protocol. The services are classified as follows:

- Authentication service – see 7.3.2.
- Messaging services – see 7.3.3.
- Directory services – see 7.3.4.

The node synchronization interface is used by the nodes to synchronize their directory data each other. MADES specifies the interface using Web services – over SOAP/HTTPS protocol – see 7.3.5.

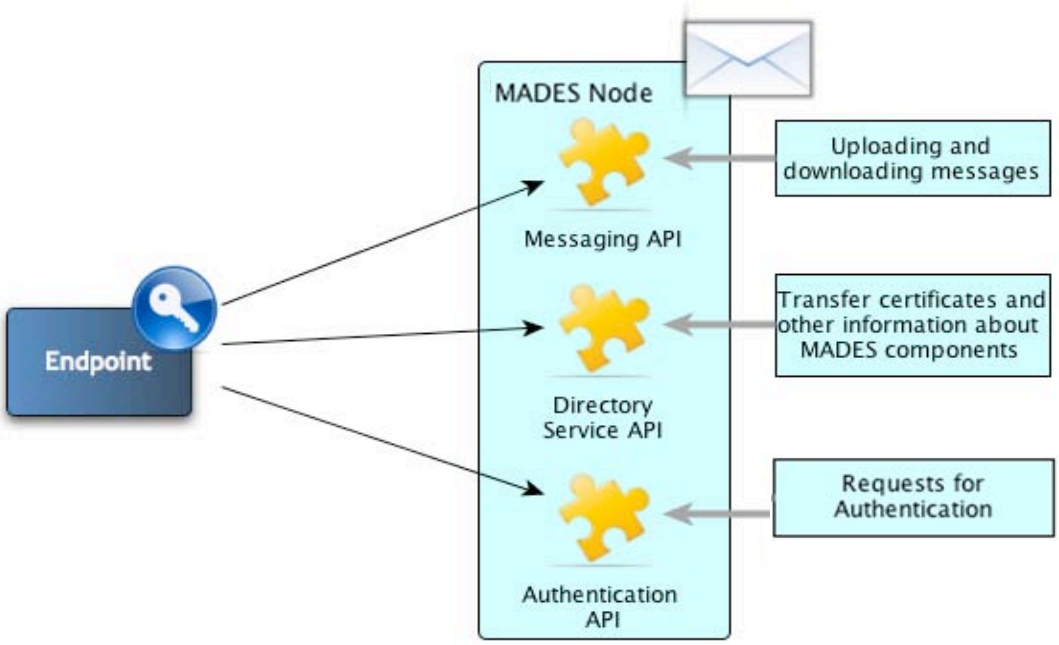

Figure 27 – Node interface – Overview

7.3.2 Authentication service

There is one authentication service, named *GetAuthenticationToken*, used by a client component to retrieve a token supplied by a server component, which is referred as the "authentication-token". The token has an expiration time (i.e. date and time). Such a token can be generated as a UUID.

The client shall then return the authentication-token signed with the authentication certificate for every following request – see 5.5.3. The client has to renew the authentication-token using the same service when expired.

Figure 28 shows the node interface for the authentication service.

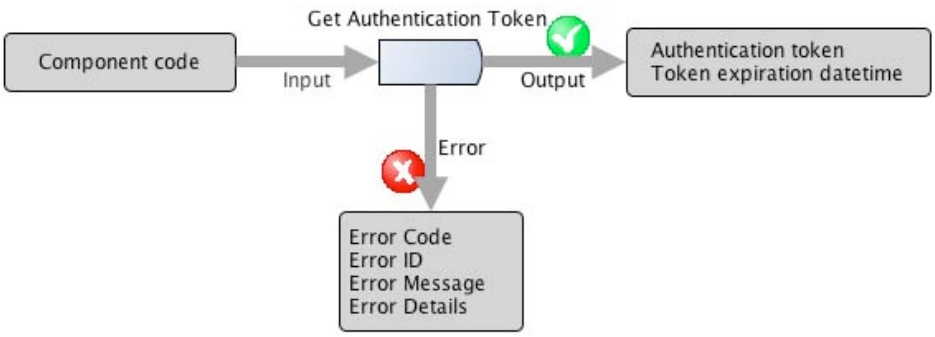

Figure 28 – Node interface – Authentication service

The service request elements are provided in Table 33.

Table 33 – Authentication – Service request elements

Element name	Type	Description	Required
componentCode	string	The component ID of the connecting client requesting for an authentication-token.	True
serviceMversion	integer	The MADES version of the current service that is requested by the client – see 6.1.3.	True

The service response elements are provided in Table 34.

Table 34 – Authentication – Service response elements

Element name	Type	Description	Required
authToken	string	The requested authentication token.	True
expiration	timestamp	The expiration date and time of the provided authentication-token.	True

7.3.3 Messaging Services

7.3.3.1 General

Messaging services are operations for bulk upload and download of messages.

The download process is a two-phase operation: first the client downloads messages from server; then it confirms that the download was successful – see 5.8.

Two limits shall be configurable within each source component regarding the bulk transfer mechanism (defined by the network governance):

- the maximum number of messages in one transfer.
- the maximum allowed size for the request (upload) or the reply (download), which contains the messages.

7.3.3.2 “Transfer confirmation” versus “acceptance”

The transfer confirmation is a technical mechanism to notify a source component that the target component has taken responsibility for the message. If the source component does not receive the confirmation, it remains responsible for the message delivery and shall then transfer it again.

The acceptance of a message by a component generally means more, i.e. that the message has passed additional validation checks. Moreover, acceptance always leads to an event notification (whether delivery or failure).

Upload: When a node accepts an uploaded message, it delegates the event notification of the event No. 3 to the sender’s endpoint and uses the upload response to do so. Other components shall not use the upload response to reject a business-message. The interpretation of the possible responses by the sender’s endpoint is:

1. No confirmation is received; the message shall be transferred again.
2. The message is accepted; a delivery acknowledgement shall be issued if the message is a business-message.
3. The message is rejected (it can only on a business-message); a failure acknowledgement shall be issued and the message shall not be transferred again.

Download: The target component of a download transfer (recipient' endpoint) always issues the acknowledgement. So there is no need to accept or reject a message when confirming the transfer (see *ConfirmDownload* 7.3.3.5).

7.3.3.3 UploadMessages service

Figure 29 shows the node interface for the UploadMessages service:

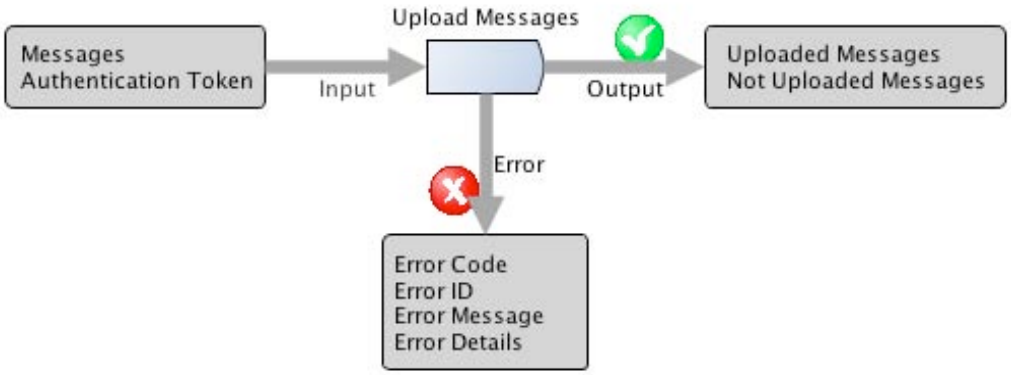

Figure 29 – Node interface – Messaging services – UploadMessages service

The service request elements are provided in Table 35.

Table 35 – UploadMessages – Service request elements

Element name	Element type	Description	Required
messages	InternalMessage[] (see Table 61)	The collection of the messages to be uploaded ordered according to priority rule in the client.	True
authToken	AuthenticationToken (see Table 53)	The authentication token provided by the server which is signed back by the client using the authentication certificate.	True
serviceMversion	integer	The MADES version of the current service that is requested by the client – see 6.1.3.	True

The service response elements are provided in Table 36.

Table 36 – UploadMessages – Service response elements

Element name	Element type	Description	Required
uploadedMessages	string[]	The collection of the IDs of the messages which are confirmed as transferred, or accepted.	False
notUploadedMessages	notUploadedMessageResponse[] (see Table 73)	The collection of {ID, error details} for each non-accepted (i.e. rejected) message.	False

The ID of every message of the request shall belong to a collection of the response.

7.3.3.4 DownloadMessages service

Figure 30 shows the node interface for the DownloadMessages service:

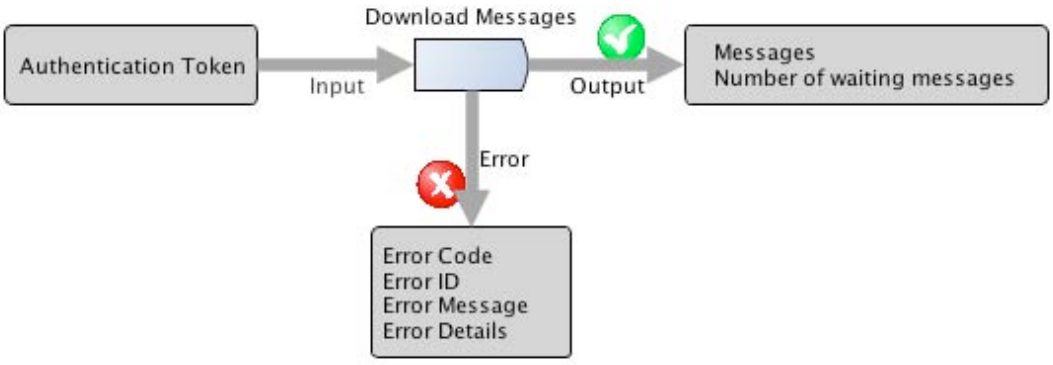

Figure 30 – Node interface – Messaging services – DownloadMessages service

A client component shall present the signed component ID of the endpoint for which it requests messages. The certificate used for signing the endpoint ID shall be the authentication certificate of the endpoint.

The node shall verify that the endpoint ID is successfully signed with a non-revoked authentication certificate of the endpoint, and log an error message when the verification fails.

The service request elements are provided in Table 37.

Table 37 – DownloadMessages – Service request elements

Element name	Element type	Description	Required
endpoints	Endpoint[] (see Table 60)	A one-element collection which contains the component ID of the recipient's endpoint whose messages are requested for download.	True
authToken	AuthenticationToken (see Table 53)	The authentication token provided by the server which is signed back by the client using the authentication certificate.	True
serviceMversion	integer	The MADES version of the current service that is requested by the client – see 6.1.3.	True

The service response elements are provided in Table 38.

Table 38 – DownloadMessages – Service response elements

Element name	Element type	Description	Required
messages	InternalMessage[] (see Table 61)	The collection of the downloaded messages ordered according to priority rule in the server— see 5.8.	False
waitingMessages	integer	The number of messages, matching the request, but not included in the current response and still waiting to be downloaded by the client.	True

7.3.3.5 ConfirmDownload service

Figure 31 shows the node interface for the ConfirmDownloadMessages service:

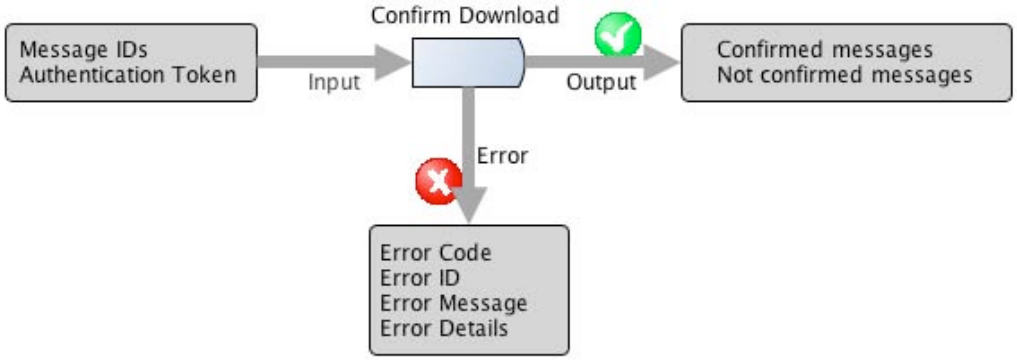

Figure 31 – Node interface – Messaging services – ConfirmDownload service

The client (source) component confirms the transfer of all or none of the messages that it previously received using a download request.

The service request elements are provided in Table 39.

Table 39 – ConfirmDownload – Service request elements

Element name	Element type	Description	Required
messageIDs	string[]	The collection of the IDs of the messages whose download transfer is confirmed.	False
authToken	AuthenticationToken (see Table 53)	The authentication token provided by the server which is signed back by the client using the authentication certificate.	True
serviceMversion	integer	The MADES version of the current service that is requested by the client – see 6.1.3.	True

The service response elements are provided in Table 40.

Table 40 – ConfirmDownload – Service response elements

Element name	Element type	Description	Required
confirmedMessages	string[]	*(Unused)*	False
notConfirmedMessages	NotConfirmedMessageResponse [] (see Table 72)	*(Unused)*	False

7.3.4 Directory services

7.3.4.1 SetComponentMversion Service

SetComponentMversion is used by a component to be accepted in the network – see 6.2.2.

To prevent that a component sends wrong data which could disrupt the network behaviour, the component ID shall be signed.

The service request elements are provided in Table 41.

Table 41 – SetComponentMversion – Service request elements

Element name	Element type	Description	Required
componentCode	string	The ID of the component requesting for network acceptance.	True
signature	string	The RSA encoding of the SHA-1 hash of the component ID (componentCode) of the component requesting for network acceptance. The certificate used for encoding shall be the authentication certificate of the component.	True
certificateID	string	The ID of the certificate used to sign, i.e. to generate the “signature”.	True
componentMVersion	integer	The installed MADES version of the component requesting for network acceptance.	True
authToken	AuthenticationToken (see Table 53)	The authentication token provided by the server which is signed back by the client using the authentication certificate.	True
serviceMversion	integer	The MADES version of the current service that is requested by the client – see 6.1.3.	True

The service response elements are provided in Table 42.

Table 42 – SetComponentMversion – Service response elements

Element name	Element type	Description	Required
nodeMversion	integer	The installed MADES version of the home node.	True
acceptance	boolean	True if the component is accepted in the network – see 6.2.2	True

7.3.4.2 GetCertificate service

GetCertificate is used to retrieve a certificate of a given type (signing, encryption, or authentication), owned by the given endpoint and possibly having the requested ID.

Figure 32 shows the node interface for the GetCertificate service:

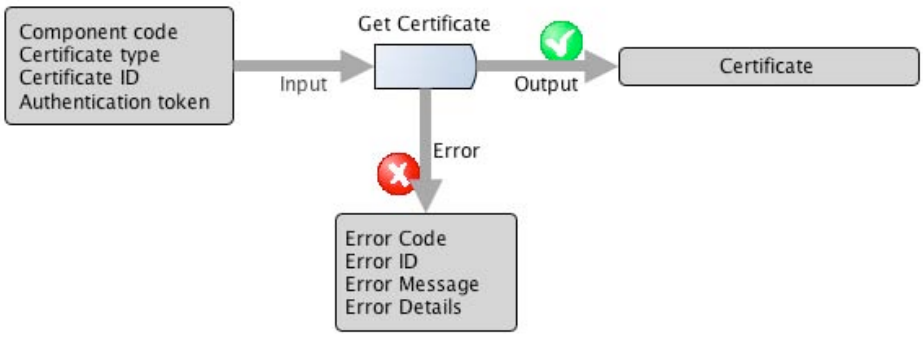

Figure 32 – Node interface – Directory services – GetCertificate service

The service request elements are provided in Table 43.

Table 43 – GetCertificate – Service request elements

Element name	Element type	Description	Required
componentCode	string	The ID of the component that owns the requested certificate.	True
type	CertificateType (see Table 54)	The type of the requested certificate,	True
certificateID	string	The ID of the requested certificate.	False
authToken	AuthenticationToken (see Table 53)	The authentication token provided by the server which is signed back by the client using the authentication certificate.	True
serviceMversion	integer	The MADES version of the current service that is requested by the client – see 6.1.3.	True

The service response elements are provided in Table 44.

Table 44 – GetCertificate – Service response elements

Element name	Element type	Description	Required
certificate	Certificate (See Table 54)	The returned certificate shall match the requested "*type*" and shall be owned by *componentCode*. If *certificateID* is also requested, the returned certificate shall also match the ID. Additional conditions about the validity and the revocation of the certificate are provided further. If no certificate matches, the response is empty.	False

Additional conditions are provided in Table 45.

Table 45 – GetCertificate – Additional conditions

Certificate type	When *certificateID* is requested	When *certificateID* is not requested
Authentication	The returned certificate shall be valid (not expired) and not revoked.	Request Error: this situation should never occur. A component shall only request for an authentication certificate to check a signed token or a signed component ID, and thus always knowing the certificate ID.
Encryption	The returned certificate can be expired or revoked, for it may be requested to decrypt a message that was composed before the expiration time or the revocation time.	The returned certificate shall be valid and not revoked. When several certificates match conditions, the service shall return the certificate that expires first.
Signing	The returned certificate can be expired or revoked, for it may be requested to check the signature of a message that was composed before the expiration time or the revocation time.	Request Error: this situation should never occur. A component shall only request for a signing certificate to check a signature and thus always knowing the certificate ID.

7.3.4.3 GetComponent service

GetComponent is used for retrieving descriptive and routing information on a component.

Figure 33 shows the node interface for the GetComponent service:

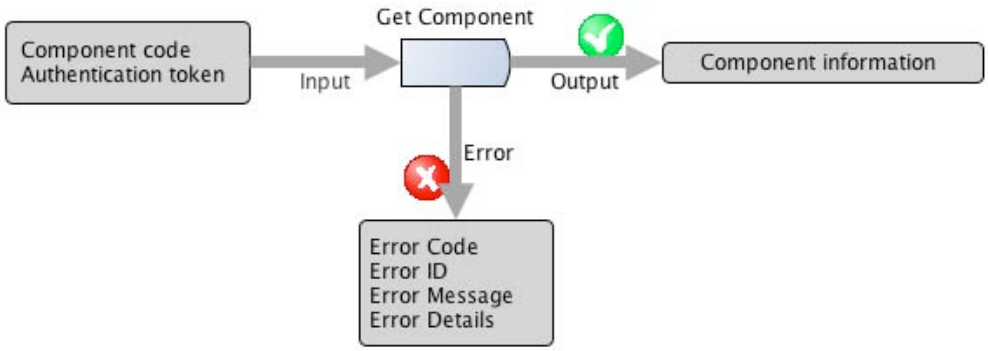

Figure 33 – Node interface – Directory services – GetComponent service

The service request elements are provided in Table 46.

Table 46 – GetComponent – Service request elements

Element name	Element type	Description	Required
componentCode	string	The ID (or code) of the requested component.	True
authToken	AuthenticationToken (see Table 53)	The authentication token provided by the server which is signed back by the client using the authentication certificate.	True
serviceMversion	integer	The MADES version of the current service that is requested by the client – see 6.1.3.	True

The service response elements are provided in Table 47.

Table 47 – GetComponent – Service response elements

Element name	Element type	Description	Required
component	ComponentInformation (see Table 58)	The directory Information about the component. If the requested component does not exist, the response is empty.	False

7.3.5 Node Synchronization interface

7.3.5.1 GetNodeMversion service

The *GetNodeMversion* service is used by a node to get the Mversion of another node – see 6.2.1.

The service request elements are provided in Table 48.

Table 48 – GetNodeMversion – Service request elements

Element name	Element type	Description	Required
mversion	integer	The installed MADES version of the requesting client node.	True

The service response elements are provided in Table 49.

Table 49 – GetNodeMversion – Service response elements

Element name	Element type	Description	Required
mversion	integer	The installed MADES version of the replying server node.	True
nodeCode	string	The component ID of the replying server node.	True

7.3.5.2 GetAllDirectoryData service

The *GetAllDirectoryData* service is used by the nodes to synchronize each other.

The service request elements are provided in Table 50.

Table 50 – GetAllDirectoryData – Service request elements

Element name	Element type	Description	Required
dversion	integer	The version of the directory data of the server node that the client node already owns. NOTE No version shall be provided if the client synchronizes for the first time with the server.	False
serviceMversion	integer	The MADES version of the current service that is requested by the client – see 6.1.3.	True

The service response elements are provided in Table 51.

Table 51 – GetAllDirectoryData – Service response elements

Element name	Element type	Description	Required
dversion	integer	The current version of the directory reference data of the replying server node.	True
nodeCode	string	The component ID of the replying server node.	True
components	ComponentDescription[] (see Table 57)	The collection of the descriptions of all components registered to the replying server node, plus the description of the node itself. The collection shall only be provided when the current directory version of the server node is strictly higher than the version already owned by the client node.	False

7.4 Format of the node-list file

The node-list file shall be in UTF-8 encoding.

The records in the file shall be delimited by the new line character (LF U+000A).

Each record provides a list of attributes for one node. Attributes are delimited by the empty space character (SPACE U+0020) and shall appear in the order given in Table 52.

Table 52 – Node attributes ordered list

Attribute	Description	Required
Node component ID	The component ID of the node	True
Primary node URL	The primary URL to access the node, formatted according to RFC 1738	True
Secondary node URL	The secondary URL to access the node, formatted according to RFC 1738	False

7.5 Typed Elements used by the interfaces

MTOM (*Message Transmission Optimization Mechanism*) is a W3C recommendation for handling binary data in SOAP messages – http://www.w3.org/TR/soap12-mtom/

MTOM shall be used to optimise the size of the messages sent. Binary data in the SOAP message have to be encoded as text because SOAP uses XML. The base64 text encoding increases the size of the data by about 33 %. MTOM provides a way to send the binary data in the original binary form. MTOM optimizes the element content that is in the canonical lexical representation of the *xsd:base64Binary* type.

All element types used in Clause 7 in the interfaces of the services are gathered from Table 53 to Table 76.

Table 53 – AuthenticationToken

Element name	Element type	Description	Required
token	string	The token received by the client (or requesting) component when it authenticated against the server (see the *GetAuthenticationToken* service).	True
signature	string	The signed token, i.e. the RSA encoding of the SHA-1 hash of the token. The encoding certificate shall be the authentication certificate of the client.	True
certificateID	string	The ID of the certificate used for signing the token.	True

Table 54 – Certificate

Element name	Element type	Description	Required
certificateID	string	The ID of the certificate.	True
certificate	base64Binary	The binary data of the certificate in DER (Distinguished Encoding Rules) format.	True
expiration	timestamp	The cache expiration date-time of the certificate if cached by client – see 5.16.5. *Do not confuse with the expiration date of the certificate as defined by the certificate issuer and included in the certificate itself.* *The element is required for directory service, but not for directory synchronization*	

Table 55 – CertificateType – string enumeration

String value	Description
AUTHENTICATION	A certificate used for TLS and token authentication
ENCRYPTION	A certificate used for encryption
SIGNING	A certificate used for signing

Table 56 – ComponentCertificate

Element name	Element type	Description	Required
type	CertificateType (see Table 55)	The type of the certificate (e.g. encryption, signing, authentication)	True
revoked	boolean	'true' if the certificate has been revoked.	False
certificate	Certificate (see Table 54)	The certificate.	True

Table 57 – ComponentDescription

Element name	Element type	Description	Required
information	ComponentInformation (see Table 58)	All about the component: ID, type, contact information, routing information.	True
certificates	ComponentCertificate[] (see Table 56)	The collection of the certificates owned by the component, whatever type (signing, encryption and authentication), and possibly more than one for some types.	True

Table 58 – ComponentInformation

Element name	Element type	Description	Required
code	string	The component ID to which is associated all information of the current data structure.	True
type	ComponentType (see Table 59)	The type of component.	True
organization	string	The organization responsible for the component.	True
person	string	The technical contact person.	True
email	string	The email of the technical contact person.	True
phone	string	The phone number of the technical contact person.	True
routing	RoutingInformation (see Table 75)	The routing information to access to the component (ex: URLs)	True
expiration	timestamp	The cache expiration date-time of the information if cached by client – see 5.16.5. *The element is required for directory service, but not for directory synchronization*	
codeMversion	integer	The MADES version to which the component complies – see 6.1.2. *NOTE Information should be initialized in thehome node at registration time. Otherwise it remains unknown until the component first connects to the network, and no message can be sent to the component.*	False

Table 59 – ComponentType – string enumeration

String value	Description
NODE	The component is a node.
ENDPOINT	The component is an endpoint.

Table 60 – Endpoint

Element name	Element type	Description	Required
code	string	The component ID of an endpoint.	True
signature	string	The RSA encoding of the SHA-1 hash of the component ID of the endpoint. The certificate used for encoding is the authentication certificate of the endpoint.	True
certificateID	string	The ID of the certificate used to encode the signature.	True

Table 61 – InternalMessage

Element name	Element type	Description	Required
messageID	string	The ID of the message.	True
receiverCode	string	*Business-message, Tracing-message* → The component ID of the recipient's endpoint. *Acknowledgement* → The *senderCode* of the original message.	True
businessType	string	*Business-message* → The business-type as provided by the sender's BA. *Tracing-message* → Irrelevant, but at least one character needed. *Acknowledgement* → The business-type of the original message.	True
content	base64Binary	*Business-message* → The encrypted content of the message, possibly compressed before encrypted. *Tracing-message* → A non empty (at least one character) irrelevant and not compressed but encrypted content. *Acknowledgement* → see 5.10.4.	True
extension	string	*Business-message* → The file extension for the document – only used if the content was transferred to the sender's endpoint through a file and by the FSSF interface (see 7.2.3). *Acknowledgement, Tracing-message* → Not used.	False
generated	dateTime	*Business-message, Tracing-message* → The date and time when the message was created by the sender's endpoint. *Acknowledgement* → The date and time of the notified event, i.e. when the acknowledgement was created in the sending component.	True
expirationTime	timestamp	*Business-message, Tracing-message* → The expiration date and time of the message – set by the sender's endpoint when accepting the message (see 5.9). *Acknowledgement* → The *expirationTime* of the original message.	True
senderCode	string	*Business-message, Tracing-message* → The component ID of the sender's endpoint. *Acknowledgement* → The ID of the component sending the acknowledgement.	True
senderDescription	string	The display name of the *senderCode* component.	True
internalType	InternalMessageType (see Table 62)	The technical type of the message.	True
relatedMessageID	string	*Business-message, Tracing-message* → Not used. *Acknowledgement* → The message ID of the original message..	False
SenderApplication	string	*Business-message, Tracing-message* → The identifier of the sender's BA as provided when sending the document. *Acknowledgement* → Not used.	False
baMessageID	string	*Business-message, Tracing-message* → An identifier of the document as provided by the sender's BA. *Acknowledgement* → Not used.	False

Element name	Element type	Description	Required
metadata	MessageMetadata (see Table 63)	The metadata added to the message by compression, signature or encryption – see 5.14.	False
messageMversion	integer	The MADES version to which the message complies – see 6.1.4	True
NOTE The underlined attributes are those included in the manifest used to generate the message signature – see 5.14.3. The "message header" refers to the set of all elements except "content".			

Table 62 – InternalMessageType – string enumeration

String Value	Description
STANDARD_MESSAGE	A business-message but not a tracing-message.
DELIVERY_ACKNOWLEDGEMENT	An acknowledgement notifying that the original STANDARD_MESSAGE has been accepted by a component.
RECEIVE_ACKNOWLEDGEMENT	An acknowledgement notifying that the original STANDARD_MESSAGE has been transferred to a recipient's BA.
FAILURE_ACKNOWLEDGEMENT	A failure-acknowledgement.
TRACING_MESSAGE	A tracing-message – see 5.12.
TRACING_ACKNOWLEDGEMENT	An acknowledgement notifying that the original TRACING_MESSAGE has been accepted by a component.

Table 63 – MessageMetadata

Element name	Element type	Description	Required
messageProcessors	MessageProcessor[]	A collection of metadata, each from a used message processor (collection count may range from 1 to 3).	False

Table 64 – MessageProcessor

Element name	Element type	Description	Required
processorID	string	The unique ID of the message processor. There are 3 processors whose IDs are: "signature" "encryption" "compressor"	True
processorData	Map	A collection of named values.	True

Table 65 – Map

Element name	Element type	Description	Required
entries	MapEntry[]	A collection of data, each provided with a name and a value, i.e. a set composed of a key (name), a type (format) and a value (according to the type).	False

Table 66 – MapEntry

Element name	Element type	Description	Required
key	string	The name of the metadata	True
type	ValueType	The type/format of the metadata.	True
value	string	The value of the metadata.	True

Table 67 – ValueType (enumeration)

String Value	Description
STRING	String
LONG	A 64bit number expressed as string. Example number 42 is represented as string "42" (without quotes)
BYTE_ARRAY	⇔ base64Binary type
BOOLEAN	A string equal to “true” or “false”.

Table 68 – MessageState (string enumeration)

String value	Description
VERIFYING	The acceptance of the message by the sender’s endpoint is pending due to connectivity problem between the sender’s endpoint and the directory services.
ACCEPTED	The message has been accepted by the sender’s endpoint.
TRANSPORTED	The message has been accepted by an intermediate component (except the recipient’s endpoint or a recipient’s BA).
DELIVERED	The message has been accepted by the recipient’s endpoint.
RECEIVED	The message has been accepted by a recipient’s BA.
FAILED	The processing of the message has failed and the delivery is stopped.

Table 69 – MessageStatus

Element name	Element type	Description
messageID	string	The UUID (Universal Unique ID) of the message whose status is reported in this data structure – see 5.2.
state	MessageState (see Table 68)	The delivery-status of the requested message. (see values in 5.4 - uppercase)
receiverCode	string	The component ID of the recipient’s endpoint of the message.
senderCode	string	The component ID of the sender’s endpoint of the message.
businessType	string	The business-type of the message.
senderApplication	string	The identifier of sender’s BA, if any and as provided by the sender’s BA in the *SendMessage* service.
baMessageID	string	The identifier of the message assigned by the sending BA, if any and as provided by the sender’s BA in the *SendMessage* service.
sendTimestamp	dateTime	The time when the message was created by the sender’s endpoint (The *generated* element of the *InternalMessage* type – see Table 61).
receiveTimestamp	dateTime	The “reception time” of the message in the sender’s endpoint. It is the time when the message state was set to DELIVERED in the sender’s endpoint, which is also the time when the acknowledgement with the DELIVERED status (event n°6) was sent.
trace	MessageTraceItem [] (see Table 70)	The collection of the traces reporting the events about the message delivery.

Table 70 – MessageTraceItem

Element name	Element type	Description	Required
timestamp	dateTime	The date and time of the reported event.	True
state	MessageTraceState (see Table 71)	The reported event	True
component	string	The ID of the component (see 5.2) where the event happened.	True
Component description	string	The display name of the component where the event happened.	True
Details	string	The English readable details about the event.	False

Table 71 – MessageTraceState (string enumeration)

String value	Description
VERIFYING	The acceptance of the message by the sender's endpoint is pending due to connectivity problem between the sender's endpoint and the directory services. (*internal event reported by the sender's endpoint*).
ACCEPTED	The message has been accepted by the sender's endpoint. (*internal event reported by the sender's endpoint*).
TRANSPORTED	The message has been accepted by an intermediate component (except the recipient's endpoint or a recipient's BA). *(event reported using a DELIVERY_ACKNOWLEDGEMENT or a TRACING_ACKNOWLEDGEMENT – see* Table 62*)*
DELIVERED	The message has been accepted by the recipient's endpoint. *(event reported using a DELIVERY_ACKNOWLEDGEMENT or a TRACING_ACKNOWLEDGEMENT – see* Table 62*)*
RECEIVED	The message has been accepted by a recipient's BA. *(event reported using an RECEIVE_ACKNOWLEDGEMENT – see* Table 62*)*
FAILED	The processing of the message has failed and the delivery is stopped. *(internal event reported by the sender's endpoint or event reported using a FAILURE_ACKNOWLEDGEMENT – see* Table 62*)*

Table 72 – NotConfirmedMessageResponse

Element name	Element type	Description	Required
messageID	string	The ID of the message whose upload failed.	True
errorCode	string	A code representing the type of the error (e.g. validation error, unexpected error)	True
errorID	string	Unique identification of the error and for the component implementation. The error ID shall be always written in the logs.	True
errorMessage	string	An English readable text describing the error.	True
errorDetails	string	(optional) Additional English readable details about the error context.	False

Table 73 – NotUploadedMessageResponse

Element name	Element type	Description	Required
messageID	string	The ID of the message whose upload failed.	True
fatal	boolean	Set to true if the error is not recoverable. The message shall then be set in the failed state by the client source component, which shall never try to upload it again.	True
businessErrorMessage	string	An English readable description of the error.	False
errorCode	string	A code representing the type of the error (e.g. validation error, unexpected error)	True
errorID	string	Unique identification of the error and for the component implementation. The error ID shall be always written in the logs.	True
errorMessage	string	An English readable text describing the error.	True
errorDetails	string	(optional) Additional English readable details about the error context.	False

Table 74 – ReceivedMessage

Element name	Element type	Description
messageID	string	The UUID (Universal Unique ID) of the message – see 5.2.
receiverCode	string	The component ID of the recipient's endpoint of the message – see 5.2.
senderCode	string	The component ID of the sender's endpoint of the message – see 5.2.
businessType	string	The business-type of the message currently transferred to the BA.
content	base64Binary	The content of the message as provided by the sender's BA in the *SendMessage* service.
senderApplication	string	The identifier of sender's BA, if any and as provided by the sender's BA in the *SendMessage* service.
baMessageID	string	The identifier of the message assigned by the sending BA, if any and as provided by the sender's BA in the *SendMessage* service.

Table 75 – RoutingInformation

Element name	Element type	Description	Required
node	string	The component ID of the component's home node.	True
primaryURL	string	The primary URL of the node according to RFC 1738	True
secondaryURL	string	The secondary URL of the node according to RFC 1738	False
nodeMversion	integer	The installed MADES version of the component's home node – see 6.1.2	True

Table 76 – SentMessage

Element name	Element type	Description	Required
receiverCode	string	The component ID of the recipient's endpoint (see 5.2) – Pattern: [A-Za-z0-9-@]+	True
businessType	string	The business-type of the message (see 5.3) – Pattern: [A-Za-z0-9]+	True

Element name	Element type	Description	Required
baMessageID	string	An identifier of the document provided by the sender's BA. Information is transported "as is" to the recipient's BA – Pattern: [A-Za-z0-9]*	False
senderApplication	string	The identifier of the sender's BA. Information is transported "as is" to the recipient's BA – Pattern: [A-Za-z0-9]*	False
content	base64Binary	The content of the message, i.e. the business document. NOTE There is no constraint about the structure of the document which is processed as a stream of bytes. E.g. it can be a human-readable XML document, multiples files compressed in ZIP format.	True

7.6 Description of the services

7.6.1 About WSDL and SOAP

The services are described using the Web Services Description Language (WSDL) 1.1[11]. See http://www.w3.org/TR/wsdl and Figure 34.

The SOAP 1.1 and SOAP 1.2 bindings allow using the interfaces via SOAP 1.1 and SOAP 1.2 protocols.

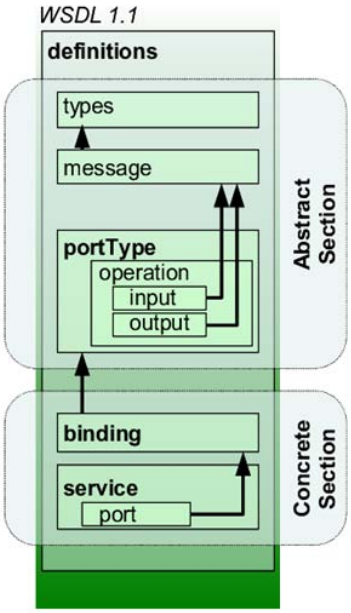

Figure 34 – WSDL 1.1 definitions

7.6.2 Endpoint interface

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="MadesEndpoint" targetNamespace="http://mades.entsoe.eu/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:ecp="http://mades.entsoe.eu/"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/">

  <wsdl:types>
    <xsd:schema targetNamespace="http://mades.entsoe.eu/">
```

11 Figure 34 from http://en.wikipedia.org/wiki/Web_Services_Description_Language.

```
      <xsd:element name="SendMessageRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="message" type="mades:SentMessage"/>
            <xsd:element minOccurs="0" name="conversationID" nillable="true"
type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="SendMessageResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="messageID" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="SendMessageError">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="errorCode" type="xsd:string"/>
            <xsd:element name="errorID" type="xsd:string"/>
            <xsd:element name="errorMessage" type="xsd:string"/>
            <xsd:element minOccurs="0" name="receiverCode" nillable="true"
type="xsd:string"/>
            <xsd:element minOccurs="0" name="errorDetails" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="ReceiveMessageRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="businessType" type="xsd:string"/>
            <xsd:element name="downloadMessage" type="xsd:boolean"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="ReceiveMessageResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0" name="receivedMessage" nillable="true"
type="mades:ReceivedMessage"/>
            <xsd:element name="remainingMessagesCount" type="xsd:long"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="ReceiveMessageError">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="errorCode" type="xsd:string"/>
            <xsd:element name="errorID" type="xsd:string"/>
            <xsd:element name="errorMessage" type="xsd:string"/>
            <xsd:element minOccurs="0" name="businessType" nillable="true"
type="xsd:string"/>
            <xsd:element minOccurs="0" name="errorDetails" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="ConfirmReceiveMessageRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="messageID" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
```

```
      </xsd:element>

      <xsd:element name="ConfirmReceiveMessageResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="messageID" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="ConfirmReceiveMessageError">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="errorCode" type="xsd:string"/>
            <xsd:element name="errorID" type="xsd:string"/>
            <xsd:element name="errorMessage" type="xsd:string"/>
            <xsd:element minOccurs="0" name="messageID" nillable="true"
type="xsd:string"/>
            <xsd:element minOccurs="0" name="errorDetails" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:complexType name="SentMessage">
        <xsd:sequence>
          <xsd:element name="receiverCode" type="xsd:string"/>
          <xsd:element name="businessType" type="xsd:string"/>
          <xsd:element name="content" type="xsd:base64Binary"/>
          <xsd:element minOccurs="0" name="senderApplication" nillable="true"
type="xsd:string"/>
          <xsd:element minOccurs="0" name="baMessageID" nillable="true"
type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="ReceivedMessage">
        <xsd:sequence>
          <xsd:element name="messageID" type="xsd:string"/>
          <xsd:element name="receiverCode" type="xsd:string"/>
          <xsd:element name="senderCode" type="xsd:string"/>
          <xsd:element name="businessType" type="xsd:string"/>
          <xsd:element name="content" type="xsd:base64Binary"/>
          <xsd:element minOccurs="0" name="senderApplication" nillable="true"
type="xsd:string"/>
          <xsd:element minOccurs="0" name="baMessageID" nillable="true"
type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="MessageStatus">
        <xsd:sequence>
          <xsd:element name="messageID" type="xsd:string"/>
          <xsd:element name="state" type="mades:MessageState"/>
          <xsd:element name="receiverCode" type="xsd:string"/>
          <xsd:element name="senderCode" type="xsd:string"/>
          <xsd:element name="businessType" type="xsd:string"/>
          <xsd:element minOccurs="0" name="senderApplication" nillable="true"
type="xsd:string"/>
          <xsd:element minOccurs="0" name="baMessageID" nillable="true"
type="xsd:string"/>
          <xsd:element name="sendTimestamp" type="xsd:dateTime"/>
          <xsd:element minOccurs="0" name="receiveTimestamp" nillable="true"
type="xsd:dateTime"/>
          <xsd:element name="trace" nillable="true" type="mades:MessageTrace"/>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="MessageTrace">
        <xsd:sequence>
```

```
          <xsd:element maxOccurs="unbounded" name="trace"
type="mades:MessageTraceItem"/>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="MessageTraceItem">
        <xsd:sequence>
          <xsd:element name="timestamp" type="xsd:dateTime"/>
          <xsd:element name="state" type="mades:MessageTraceState"/>
          <xsd:element name="component" type="xsd:string"/>
          <xsd:element name="componentDescription" type="xsd:string"/>
          <xsd:element name="details" nillable="true" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:element name="ConnectivityTestRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="receiverCode" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="ConnectivityTestResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="messageID" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="ConnectivityTestError">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="errorCode" type="xsd:string"/>
            <xsd:element name="errorID" type="xsd:string"/>
            <xsd:element name="errorMessage" type="xsd:string"/>
            <xsd:element minOccurs="0" name="receiverCode" nillable="true"
type="xsd:string"/>
            <xsd:element minOccurs="0" name="errorDetails" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="CheckMessageStatusRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="messageID" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="CheckMessageStatusResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="messageStatus" type="mades:MessageStatus"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="CheckMessageStatusError">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="errorCode" type="xsd:string"/>
            <xsd:element name="errorID" type="xsd:string"/>
            <xsd:element name="errorMessage" type="xsd:string"/>
            <xsd:element minOccurs="0" name="messageID" nillable="true"
type="xsd:string"/>
            <xsd:element minOccurs="0" name="errorDetails" type="xsd:string"/>
          </xsd:sequence>
```

```
      </xsd:complexType>
    </xsd:element>

    <xsd:simpleType name="MessageState">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="VERIFYING"/>
        <xsd:enumeration value="ACCEPTED"/>
        <xsd:enumeration value="DELIVERING"/>
        <xsd:enumeration value="DELIVERED"/>
        <xsd:enumeration value="RECEIVED"/>
        <xsd:enumeration value="FAILED"/>
      </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="MessageTraceState">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="VERIFYING"/>
        <xsd:enumeration value="ACCEPTED"/>
        <xsd:enumeration value="TRANSPORTED"/>
        <xsd:enumeration value="DELIVERED"/>
        <xsd:enumeration value="RECEIVED"/>
        <xsd:enumeration value="FAILED"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:schema>
</wsdl:types>

<wsdl:message name="SendMessageRequest">
  <wsdl:part name="parameters" element="mades:SendMessageRequest"/>
</wsdl:message>

<wsdl:message name="SendMessageResponse">
  <wsdl:part name="parameters" element="mades:SendMessageResponse"/>
</wsdl:message>

<wsdl:message name="ConnectivityTestFault">
  <wsdl:part name="fault" element="mades:ConnectivityTestError"/>
</wsdl:message>

<wsdl:message name="ReceiveMessageRequest">
  <wsdl:part name="parameters" element="mades:ReceiveMessageRequest"/>
</wsdl:message>

<wsdl:message name="ConfirmReceiveMessageRequest">
  <wsdl:part name="parameters" element="mades:ConfirmReceiveMessageRequest"/>
</wsdl:message>

<wsdl:message name="ConnectivityTestRequest">
  <wsdl:part name="parameters" element="mades:ConnectivityTestRequest"/>
</wsdl:message>

<wsdl:message name="CheckMessageStatusResponse">
  <wsdl:part name="parameters" element="mades:CheckMessageStatusResponse"/>
</wsdl:message>

<wsdl:message name="ConfirmReceiveMessageResponse">
  <wsdl:part name="parameters" element="mades:ConfirmReceiveMessageResponse"/>
</wsdl:message>

<wsdl:message name="ReceiveMessageFault">
  <wsdl:part name="fault" element="mades:ReceiveMessageError"/>
</wsdl:message>

<wsdl:message name="CheckMessageStatusFault">
  <wsdl:part name="fault" element="mades:CheckMessageStatusError"/>
</wsdl:message>

<wsdl:message name="CheckMessageStatusRequest">
  <wsdl:part name="parameters" element="mades:CheckMessageStatusRequest"/>
</wsdl:message>
```

```
  <wsdl:message name="ConfirmReceiveMessageFault">
    <wsdl:part name="fault" element="mades:ConfirmReceiveMessageError"/>
  </wsdl:message>

  <wsdl:message name="SendMessageFault">
    <wsdl:part name="fault" element="mades:SendMessageError"/>
  </wsdl:message>

  <wsdl:message name="ReceiveMessageResponse">
    <wsdl:part name="parameters" element="mades:ReceiveMessageResponse"/>
  </wsdl:message>

  <wsdl:message name="ConnectivityTestResponse">
    <wsdl:part name="parameters" element="mades:ConnectivityTestResponse"/>
  </wsdl:message>

  <wsdl:portType name="MadesEndpoint">
    <wsdl:operation name="SendMessage">
      <wsdl:input message="mades:SendMessageRequest"/>
      <wsdl:output message="mades:SendMessageResponse"/>
      <wsdl:fault name="SendMessageError" message="mades:SendMessageFault"/>
    </wsdl:operation>
    <wsdl:operation name="ReceiveMessage">
      <wsdl:input message="mades:ReceiveMessageRequest"/>
      <wsdl:output message="mades:ReceiveMessageResponse"/>
      <wsdl:fault name="ReceiveMessageError" message="mades:ReceiveMessageFault"/>
    </wsdl:operation>
    <wsdl:operation name="ConfirmReceiveMessage">
      <wsdl:input message="mades:ConfirmReceiveMessageRequest"/>
      <wsdl:output message="mades:ConfirmReceiveMessageResponse"/>
      <wsdl:fault name="ConfirmReceiveMessageError"
message="mades:ConfirmReceiveMessageFault"/>
    </wsdl:operation>
    <wsdl:operation name="ConnectivityTest">
      <wsdl:input message="mades:ConnectivityTestRequest"/>
      <wsdl:output message="mades:ConnectivityTestResponse"/>
      <wsdl:fault name="ConnectivityTestError"
message="mades:ConnectivityTestFault"/>
    </wsdl:operation>
    <wsdl:operation name="CheckMessageStatus">
      <wsdl:input message="mades:CheckMessageStatusRequest"/>
      <wsdl:output message="mades:CheckMessageStatusResponse"/>
      <wsdl:fault name="CheckMessageStatusError"
message="mades:CheckMessageStatusFault"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="MadesEndpointSOAP12" type="mades:MadesEndpoint">
    <soap12:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="SendMessage">
      <soap12:operation soapAction="http://mades.entsoe.eu/SendMessage"/>
      <wsdl:input> <soap12:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap12:body use="literal"/> </wsdl:output>
      <wsdl:fault name="SendMessageError"> <soap12:fault name="SendMessageError"
use="literal"/> </wsdl:fault>
    </wsdl:operation>
    <wsdl:operation name="ReceiveMessage">
      <soap12:operation soapAction="http://mades.entsoe.eu/ReceiveMessage"/>
      <wsdl:input> <soap12:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap12:body use="literal"/> </wsdl:output>
      <wsdl:fault name="ReceiveMessageError"> <soap12:fault
name="ReceiveMessageError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
    <wsdl:operation name="ConfirmReceiveMessage">
      <soap12:operation soapAction="http://mades.entsoe.eu/ConfirmReceiveMessage"/>
      <wsdl:input> <soap12:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap12:body use="literal"/> </wsdl:output>
```

```
      <wsdl:fault name="ConfirmReceiveMessageError"> <soap12:fault
name="ConfirmReceiveMessageError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
    <wsdl:operation name="ConnectivityTest">
      <soap12:operation soapAction="http://mades.entsoe.eu/ConnectivityTest"/>
      <wsdl:input> <soap12:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap12:body use="literal"/> </wsdl:output>
      <wsdl:fault name="ConnectivityTestError"> <soap12:fault
name="ConnectivityTestError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
    <wsdl:operation name="CheckMessageStatus">
      <soap12:operation soapAction="http://mades.entsoe.eu/CheckMessageStatus"/>
      <wsdl:input> <soap12:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap12:body use="literal"/> </wsdl:output>
      <wsdl:fault name="CheckMessageStatusError"> <soap12:fault
name="CheckMessageStatusError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:binding name="MadesEndpointSOAP11" type="mades:MadesEndpoint">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="SendMessage">
      <soap:operation soapAction="http://mades.entsoe.eu/SendMessage"/>
      <wsdl:input> <soap:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap:body use="literal"/> </wsdl:output>
      <wsdl:fault name="SendMessageError"> <soap:fault name="SendMessageError"
use="literal"/> </wsdl:fault>
    </wsdl:operation>
    <wsdl:operation name="ReceiveMessage">
      <soap:operation soapAction="http://mades.entsoe.eu/ReceiveMessage"/>
      <wsdl:input> <soap:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap:body use="literal"/> </wsdl:output>
      <wsdl:fault name="ReceiveMessageError"> <soap:fault name="ReceiveMessageError"
use="literal"/> </wsdl:fault>
    </wsdl:operation>
    <wsdl:operation name="ConfirmReceiveMessage">
      <soap:operation soapAction="http://mades.entsoe.eu/ConfirmReceiveMessage"/>
      <wsdl:input> <soap:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap:body use="literal"/> </wsdl:output>
      <wsdl:fault name="ConfirmReceiveMessageError"> <soap:fault
name="ConfirmReceiveMessageError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
    <wsdl:operation name="ConnectivityTest">
      <soap:operation soapAction="http://mades.entsoe.eu/ConnectivityTest"/>
      <wsdl:input> <soap:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap:body use="literal"/> </wsdl:output>
      <wsdl:fault name="ConnectivityTestError"> <soap:fault
name="ConnectivityTestError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
    <wsdl:operation name="CheckMessageStatus">
      <soap:operation soapAction="http://mades.entsoe.eu/CheckMessageStatus"/>
      <wsdl:input> <soap:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap:body use="literal"/> </wsdl:output>
      <wsdl:fault name="CheckMessageStatusError"> <soap:fault
name="CheckMessageStatusError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="MadesEndpointService">
    <wsdl:port name="MadesEndpointSOAP12" binding="mades:MadesEndpointSOAP12">
      <soap12:address location="http://mades.entsoe.eu"/>
    </wsdl:port>
    <wsdl:port name="MadesEndpointSOAP11" binding="mades:MadesEndpointSOAP11">
      <soap:address location="http://mades.entsoe.eu"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

7.6.3 Node interface

7.6.3.1 Authentication service

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="MadesAuthenticationService"
targetNamespace="http://mades.entsoe.eu/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:ecp="http://mades.entsoe.eu/"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/">

  <wsdl:types>
    <xsd:schema targetNamespace="http://mades.entsoe.eu/">

      <xsd:element name="GetAuthenticationTokenRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="componentCode" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="GetAuthenticationTokenResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="authToken" type="xsd:string"/>
            <xsd:element name="expiration" type="xsd:long"/>
            <xsd:element minOccurs="0" name="serviceMversion" nillable="true"
type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="GetAuthenticationTokenError">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="errorCode" type="xsd:string"/>
            <xsd:element name="errorID" type="xsd:string"/>
            <xsd:element name="errorMessage" type="xsd:string"/>
            <xsd:element minOccurs="0" name="errorDetails" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </wsdl:types>

  <wsdl:message name="GetAuthenticationTokenResponse">
    <wsdl:part name="parameters" element="mades:GetAuthenticationTokenResponse"/>
  </wsdl:message>

  <wsdl:message name="GetAuthenticationTokenFault">
    <wsdl:part name="fault" element="mades:GetAuthenticationTokenError"/>
  </wsdl:message>

  <wsdl:message name="GetAuthenticationTokenRequest">
    <wsdl:part name="parameters" element="mades:GetAuthenticationTokenRequest"/>
  </wsdl:message>

  <wsdl:portType name="MadesAuthenticationService">
    <wsdl:operation name="GetAuthenticationToken">
      <wsdl:input message="mades:GetAuthenticationTokenRequest"/>
      <wsdl:output message="mades:GetAuthenticationTokenResponse"/>
      <wsdl:fault name="GetAuthenticationTokenError"
message="mades:GetAuthenticationTokenFault"/>
    </wsdl:operation>
  </wsdl:portType>
```

```
  <wsdl:binding name="MadesAuthenticationServiceSOAP12"
type="mades:MadesAuthenticationService">
    <soap12:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="GetAuthenticationToken">
      <soap12:operation soapAction="http://mades.entsoe.eu/GetAuthenticationToken"/>
      <wsdl:input> <soap12:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap12:body use="literal"/> </wsdl:output>
      <wsdl:fault name="GetAuthenticationTokenError"> <soap12:fault
name="GetAuthenticationTokenError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:binding name="MadesAuthenticationServiceSOAP11"
type="mades:MadesAuthenticationService">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="GetAuthenticationToken">
      <soap:operation soapAction="http://mades.entsoe.eu/GetAuthenticationToken"/>
      <wsdl:input> <soap:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap:body use="literal"/> </wsdl:output>
      <wsdl:fault name="GetAuthenticationTokenError"> <soap:fault
name="GetAuthenticationTokenError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="MadesAuthenticationService">
    <wsdl:port name="MadesAuthenticationServiceSOAP12"
binding="mades:MadesAuthenticationServiceSOAP12">
      <soap12:address location="http://mades.entsoe.eu"/>
    </wsdl:port>
    <wsdl:port name="MadesAuthenticationServiceSOAP11"
binding="mades:MadesAuthenticationServiceSOAP11">
      <soap:address location="http://mades.entsoe.eu"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

7.6.3.2 Messaging services

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="MadesInternalMessaging"
targetNamespace="http://mades.entsoe.eu/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:ecp="http://mades.entsoe.eu/"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/">

  <wsdl:types>
    <xsd:schema targetNamespace="http://mades.entsoe.eu/">

      <xsd:element name="UploadMessagesRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element maxOccurs="unbounded" name="messages"
type="mades:InternalMessage"/>
            <xsd:element name="authToken" type="mades:AuthenticationToken"/>
            <xsd:element minOccurs="0" name="serviceMversion" nillable="true"
type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="UploadMessagesResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element maxOccurs="unbounded" minOccurs="0" name="uploadedMessages"
type="xsd:string"/>
```

```
            <xsd:element maxOccurs="unbounded" minOccurs="0"
name="notUploadedMessages" type="mades:NotUploadedMessageResponse"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="UploadMessagesError">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="errorCode" type="xsd:string"/>
            <xsd:element name="errorID" type="xsd:string"/>
            <xsd:element name="errorMessage" type="xsd:string"/>
            <xsd:element minOccurs="0" name="errorDetails" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="DownloadMessagesRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element maxOccurs="unbounded" name="endpoints"
type="mades:Endpoint">
            <xsd:element name="authToken" type="mades:AuthenticationToken"/>
            <xsd:element minOccurs="0" name="serviceMversion" nillable="true"
type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="DownloadMessagesResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element maxOccurs="unbounded" minOccurs="0" name="messages"
type="mades:InternalMessage"/>
            <xsd:element name="waitingMessages" type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="DownloadMessagesError">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="errorCode" type="xsd:string"/>
            <xsd:element name="errorID" type="xsd:string"/>
            <xsd:element name="errorMessage" type="xsd:string"/>
            <xsd:element minOccurs="0" name="errorDetails" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="ConfirmDownloadRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element maxOccurs="unbounded" minOccurs="0" name="messageIDs"
type="xsd:string"/>
            <xsd:element name="authToken" type="mades:AuthenticationToken"/>
            <xsd:element minOccurs="0" name="serviceMversion" nillable="true"
type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="ConfirmDownloadResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element maxOccurs="unbounded" minOccurs="0" name="confirmedMessages"
type="xsd:string"/>
            <xsd:element maxOccurs="unbounded" minOccurs="0"
name="notConfirmedMessages" type="mades:NotConfirmedMessageResponse"/>
          </xsd:sequence>
```

```
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="ConfirmDownloadError">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="errorCode" type="xsd:string"/>
            <xsd:element name="errorID" type="xsd:string"/>
            <xsd:element name="errorMessage" type="xsd:string"/>
            <xsd:element minOccurs="0" name="errorDetails" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:complexType name="InternalMessage">
        <xsd:sequence>
          <xsd:element name="messageID" type="xsd:string"/>
          <xsd:element name="receiverCode" type="xsd:string"/>
          <xsd:element name="businessType" type="xsd:string"/>
          <xsd:element name="content" type="xsd:base64Binary"/>
          <xsd:element minOccurs="0" name="extension" nillable="true"
type="xsd:string"/>
          <xsd:element name="generated" type="xsd:dateTime"/>
          <xsd:element minOccurs="0" name="expirationTime" nillable="true"
type="xsd:long"/>
          <xsd:element name="senderCode" type="xsd:string"/>
          <xsd:element name="senderDescription" type="xsd:string"/>
          <xsd:element name="internalType" type="mades:InternalMessageType"/>
          <xsd:element minOccurs="0" name="relatedMessageID" nillable="true"
type="xsd:string"/>
          <xsd:element minOccurs="0" name="senderApplication" nillable="true"
type="xsd:string"/>
          <xsd:element minOccurs="0" name="baMessageID" nillable="true"
type="xsd:string"/>
          <xsd:element name="metadata" type="mades:MessageMetadata"/>
          <xsd:element minOccurs="0" name="messageMversion" nillable="true"
type="xsd:int"/>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:simpleType name="InternalMessageType">
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="STANDARD_MESSAGE"/>
          <xsd:enumeration value="DELIVERY_ACKNOWLEDGEMENT"/>
          <xsd:enumeration value="RECEIVE_ACKNOWLEDGEMENT"/>
          <xsd:enumeration value="FAILURE_ACKNOWLEDGEMENT"/>
          <xsd:enumeration value="TRACING_MESSAGE"/>
          <xsd:enumeration value="TRACING_ACKNOWLEDGEMENT"/>
        </xsd:restriction>
      </xsd:simpleType>

      <xsd:complexType name="MessageMetadata">
        <xsd:sequence>
          <xsd:element maxOccurs="unbounded" minOccurs="0" name="messageProcessors"
type="mades:MessageProcessor"/>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="MessageProcessor">
        <xsd:sequence>
          <xsd:element name="processorID" type="xsd:string"/>
          <xsd:element name="processorData" type="mades:Map"/>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="NotUploadedMessageResponse">
        <xsd:sequence>
          <xsd:element name="messageID" type="xsd:string"/>
          <xsd:element name="fatal" type="xsd:boolean"/>
```

```
          <xsd:element minOccurs="0" name="businessErrorMessage" nillable="true"
type="xsd:string"/>
          <xsd:element name="errorCode" type="xsd:string"/>
          <xsd:element name="errorID" type="xsd:string"/>
          <xsd:element name="errorMessage" type="xsd:string"/>
          <xsd:element minOccurs="0" name="errorDetails" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="NotConfirmedMessageResponse">
        <xsd:sequence>
          <xsd:element name="messageID" type="xsd:string"/>
          <xsd:element name="errorCode" type="xsd:string"/>
          <xsd:element name="errorID" type="xsd:string"/>
          <xsd:element name="errorMessage" type="xsd:string"/>
          <xsd:element minOccurs="0" name="errorDetails" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="Endpoint">
        <xsd:sequence>
          <xsd:element name="code" type="xsd:string"/>
          <xsd:element name="signature" type="xsd:string"/>
          <xsd:element name="certificateID" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="AuthenticationToken">
        <xsd:sequence>
          <xsd:element name="token" type="xsd:string"/>
          <xsd:element name="signature" type="xsd:string"/>
          <xsd:element name="certificateID" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="Map">
        <xsd:sequence>
          <xsd:element maxOccurs="unbounded" minOccurs="0" name="entries"
type="mades:MapEntry"/>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="MapEntry">
        <xsd:sequence>
          <xsd:element name="key" type="xsd:string"/>
          <xsd:element name="type" type="mades:ValueType"/>
          <xsd:element name="value" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:simpleType name="ValueType">
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="STRING"/>
          <xsd:enumeration value="LONG"/>
          <xsd:enumeration value="BYTE_ARRAY"/>
          <xsd:enumeration value="BOOLEAN"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:schema>
  </wsdl:types>

  <wsdl:message name="ConfirmDownloadResponse">
    <wsdl:part name="parameters" element="mades:ConfirmDownloadResponse"/>
  </wsdl:message>

  <wsdl:message name="UploadMessagesFault">
    <wsdl:part name="fault" element="mades:UploadMessagesError"/>
  </wsdl:message>

  <wsdl:message name="UploadMessagesRequest">
```

```
    <wsdl:part name="parameters" element="mades:UploadMessagesRequest"/>
  </wsdl:message>

  <wsdl:message name="DownloadMessagesFault">
    <wsdl:part name="fault" element="mades:DownloadMessagesError"/>
  </wsdl:message>

  <wsdl:message name="UploadMessagesResponse">
    <wsdl:part name="parameters" element="mades:UploadMessagesResponse"/>
  </wsdl:message>

  <wsdl:message name="DownloadMessagesRequest">
    <wsdl:part name="parameters" element="mades:DownloadMessagesRequest"/>
  </wsdl:message>

  <wsdl:message name="ConfirmDownloadFault">
    <wsdl:part name="fault" element="mades:ConfirmDownloadError"/>
  </wsdl:message>

  <wsdl:message name="DownloadMessagesResponse">
    <wsdl:part name="parameters" element="mades:DownloadMessagesResponse"/>
  </wsdl:message>

  <wsdl:message name="ConfirmDownloadRequest">
    <wsdl:part name="parameters" element="mades:ConfirmDownloadRequest"/>
  </wsdl:message>

  <wsdl:portType name="MadesInternalMessaging">
    <wsdl:operation name="UploadMessages">
      <wsdl:input message="mades:UploadMessagesRequest"/>
      <wsdl:output message="mades:UploadMessagesResponse"/>
      <wsdl:fault name="UploadMessagesError" message="mades:UploadMessagesFault"/>
    </wsdl:operation>
    <wsdl:operation name="DownloadMessages">
      <wsdl:input message="mades:DownloadMessagesRequest"/>
      <wsdl:output message="mades:DownloadMessagesResponse"/>
      <wsdl:fault name="DownloadMessagesError"
message="mades:DownloadMessagesFault"/>
    </wsdl:operation>
    <wsdl:operation name="ConfirmDownload">
      <wsdl:input message="mades:ConfirmDownloadRequest"/>
      <wsdl:output message="mades:ConfirmDownloadResponse"/>
      <wsdl:fault name="ConfirmDownloadError" message="mades:ConfirmDownloadFault"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="MadesInternalMessagingSOAP11"
type="mades:MadesInternalMessaging">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="UploadMessages">
      <soap:operation soapAction="http://mades.entsoe.eu/UploadMessages"/>
      <wsdl:input> <soap:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap:body use="literal"/> </wsdl:output>
      <wsdl:fault name="UploadMessagesError"> <soap:fault name="UploadMessagesError"
use="literal"/> </wsdl:fault>
    </wsdl:operation>
    <wsdl:operation name="DownloadMessages">
      <soap:operation soapAction="http://mades.entsoe.eu/DownloadMessages"/>
      <wsdl:input> <soap:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap:body use="literal"/> </wsdl:output>
      <wsdl:fault name="DownloadMessagesError"> <soap:fault
name="DownloadMessagesError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
    <wsdl:operation name="ConfirmDownload">
      <soap:operation soapAction="http://mades.entsoe.eu/ConfirmDownload"/>
      <wsdl:input> <soap:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap:body use="literal"/> </wsdl:output>
      <wsdl:fault name="ConfirmDownloadError"> <soap:fault
name="ConfirmDownloadError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
```

```
  </wsdl:binding>

  <wsdl:binding name="MadesInternalMessagingSOAP12"
type="mades:MadesInternalMessaging">
    <soap12:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="UploadMessages">
      <soap12:operation soapAction="http://mades.entsoe.eu/UploadMessages"/>
      <wsdl:input> <soap12:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap12:body use="literal"/> </wsdl:output>
      <wsdl:fault name="UploadMessagesError"> <soap12:fault
name="UploadMessagesError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
    <wsdl:operation name="DownloadMessages">
      <soap12:operation soapAction="http://mades.entsoe.eu/DownloadMessages"/>
      <wsdl:input> <soap12:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap12:body use="literal"/> </wsdl:output>
      <wsdl:fault name="DownloadMessagesError"> <soap12:fault
name="DownloadMessagesError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
    <wsdl:operation name="ConfirmDownload">
      <soap12:operation soapAction="http://mades.entsoe.eu/ConfirmDownload"/>
      <wsdl:input> <soap12:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap12:body use="literal"/> </wsdl:output>
      <wsdl:fault name="ConfirmDownloadError"> <soap12:fault
name="ConfirmDownloadError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="MadesInternalMessagingService">
    <wsdl:port name="MadesInternalMessagingSOAP11"
binding="mades:MadesInternalMessagingSOAP11">
      <soap:address location="http://mades.entsoe.eu"/>
    </wsdl:port>
    <wsdl:port name="MadesInternalMessagingSOAP12"
binding="mades:MadesInternalMessagingSOAP12">
      <soap12:address location="http://mades.entsoe.eu"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

7.6.3.3 Directory services

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="MadesDirectoryService"
targetNamespace="http://mades.entsoe.eu/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:ecp="http://mades.entsoe.eu/"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/">

  <wsdl:types>
    <xsd:schema targetNamespace="http://mades.entsoe.eu/">

      <xsd:element name="GetCertificateRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="componentCode" type="xsd:string"/>
            <xsd:element name="type" type="mades:CertificateType"/>
            <xsd:element minOccurs="0" name="certificateID" nillable="true"
type="xsd:string"/>
            <xsd:element name="authToken" type="mades:AuthenticationToken"/>
            <xsd:element minOccurs="0" name="serviceMversion" nillable="true"
type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
```

```
      <xsd:element name="GetCertificateResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0" name="certificate" nillable="true"
type="mades:Certificate"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="GetCertificateError">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="errorCode" type="xsd:string"/>
            <xsd:element name="errorID" type="xsd:string"/>
            <xsd:element name="errorMessage" type="xsd:string"/>
            <xsd:element minOccurs="0" name="errorDetails" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="GetComponentRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="componentCode" type="xsd:string"/>
            <xsd:element name="authToken" type="mades:AuthenticationToken"/>
            <xsd:element minOccurs="0" name="serviceMversion" nillable="true"
type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="GetComponentResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0" name="component" nillable="true"
type="mades:ComponentInformation"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="GetComponentError">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="errorCode" type="xsd:string"/>
            <xsd:element name="errorID" type="xsd:string"/>
            <xsd:element name="errorMessage" type="xsd:string"/>
            <xsd:element minOccurs="0" name="errorDetails" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="SetComponentMversionRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="componentCode" type="xsd:string"/>
            <xsd:element name="signature" type="xsd:string"/>
            <xsd:element name="certificateID" type="xsd:string"/>
            <xsd:element name="componentMVersion" type="xsd:int"/>
            <xsd:element name="authToken" type="mades:AuthenticationToken"/>
            <xsd:element minOccurs="0" name="serviceMversion" nillable="true"
type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="SetComponentMversionResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="nodeMversion" type="xsd:int"/>
            <xsd:element name="acceptance" type="xsd:boolean"/>
```

```
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="SetComponentMversionError">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="errorCode" type="xsd:string"/>
        <xsd:element name="errorID" type="xsd:string"/>
        <xsd:element name="errorMessage" type="xsd:string"/>
        <xsd:element minOccurs="0" name="errorDetails" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="Certificate">
    <xsd:sequence>
      <xsd:element name="certificateID" type="xsd:string"/>
      <xsd:element name="certificate" type="xsd:base64Binary"/>
      <xsd:element name="expiration" type="xsd:long"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:simpleType name="CertificateType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="AUTHENTICATION"/>
      <xsd:enumeration value="ENCRYPTION"/>
      <xsd:enumeration value="SIGNING"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:complexType name="ComponentInformation">
    <xsd:sequence>
      <xsd:element name="code" type="xsd:string"/>
      <xsd:element name="type" type="mades:ComponentType"/>
      <xsd:element name="organization" type="xsd:string"/>
      <xsd:element name="person" type="xsd:string"/>
      <xsd:element name="email" type="xsd:string"/>
      <xsd:element name="phone" type="xsd:string"/>
      <xsd:element name="routing" type="mades:RoutingInformation"/>
      <xsd:element minOccurs="0" name="expiration" nillable="true"
type="xsd:long"/>
      <xsd:element minOccurs="0" name="codeMversion" nillable="true"
type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="RoutingInformation">
    <xsd:sequence>
      <xsd:element name="node" type="xsd:string"/>
      <xsd:element name="primaryURL" type="xsd:string"/>
      <xsd:element minOccurs="0" name="secondaryURL" nillable="true"
type="xsd:string"/>
      <xsd:element minOccurs="0" name="nodeMversion" nillable="true"
type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:simpleType name="ComponentType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="NODE"/>
      <xsd:enumeration value="ENDPOINT"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:complexType name="AuthenticationToken">
    <xsd:sequence>
      <xsd:element name="token" type="xsd:string"/>
      <xsd:element name="signature" type="xsd:string"/>
      <xsd:element name="certificateID" type="xsd:string"/>
```

```
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>
  </wsdl:types>

  <wsdl:message name="GetComponentRequest">
    <wsdl:part name="parameters" element="mades:GetComponentRequest"/>
  </wsdl:message>

  <wsdl:message name="GetCertificateResponse">
    <wsdl:part name="parameters" element="mades:GetCertificateResponse"/>
  </wsdl:message>

  <wsdl:message name="GetComponentResponse">
    <wsdl:part name="parameters" element="mades:GetComponentResponse"/>
  </wsdl:message>

  <wsdl:message name="GetCertificateRequest">
    <wsdl:part name="parameters" element="mades:GetCertificateRequest"/>
  </wsdl:message>

  <wsdl:message name="GetCertificateFault">
    <wsdl:part name="fault" element="mades:GetCertificateError"/>
  </wsdl:message>

  <wsdl:message name="GetComponentFault">
    <wsdl:part name="fault" element="mades:GetComponentError"/>
  </wsdl:message>

  <wsdl:message name="SetComponentMversionRequest">
    <wsdl:part name="parameters" element="mades:SetComponentMversionRequest"/>
  </wsdl:message>

  <wsdl:message name="SetComponentMversionResponse">
    <wsdl:part name="parameters" element="mades:SetComponentMversionResponse"/>
  </wsdl:message>

  <wsdl:message name="SetComponentMversionFault">
    <wsdl:part name="fault" element="mades:SetComponentMversionError"/>
  </wsdl:message>

  <wsdl:portType name="MadesDirectoryService">
    <wsdl:operation name="GetCertificate">
      <wsdl:input message="mades:GetCertificateRequest"/>
      <wsdl:output message="mades:GetCertificateResponse"/>
      <wsdl:fault name="GetCertificateError" message="mades:GetCertificateFault"/>
    </wsdl:operation>
    <wsdl:operation name="GetComponent">
      <wsdl:input message="mades:GetComponentRequest"/>
      <wsdl:output message="mades:GetComponentResponse"/>
      <wsdl:fault name="GetComponentError" message="mades:GetComponentFault"/>
    </wsdl:operation>
    <wsdl:operation name="SetComponentMversion">
      <wsdl:input message="mades:SetComponentMversionRequest"/>
      <wsdl:output message="mades:SetComponentMversionResponse"/>
      <wsdl:fault name="SetComponentMversionError"
message="mades:SetComponentMversionFault"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="MadesDirectoryServiceSOAP11"
type="mades:MadesDirectoryService">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="GetCertificate">
      <soap:operation soapAction="http://mades.entsoe.eu/GetCertificate"/>
      <wsdl:input> <soap:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap:body use="literal"/> </wsdl:output>
      <wsdl:fault name="GetCertificateError"> <soap:fault name="GetCertificateError"
use="literal"/> </wsdl:fault>
    </wsdl:operation>
```

```
    <wsdl:operation name="GetComponent">
      <soap:operation soapAction="http://mades.entsoe.eu/GetComponent"/>
      <wsdl:input> <soap:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap:body use="literal"/> </wsdl:output>
      <wsdl:fault name="GetComponentError"> <soap:fault name="GetComponentError"
use="literal"/> </wsdl:fault>
    </wsdl:operation>
    <wsdl:operation name="SetComponentMversion">
      <soap:operation soapAction="http://mades.entsoe.eu/SetComponentMversion"/>
      <wsdl:input> <soap:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap:body use="literal"/> </wsdl:output>
      <wsdl:fault name="SetComponentMversionError"> <soap:fault
name="SetComponentMversionError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:binding name="MadesDirectoryServiceSOAP12"
type="mades:MadesDirectoryService">
    <soap12:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="GetCertificate">
      <soap12:operation soapAction="http://mades.entsoe.eu/GetCertificate"/>
      <wsdl:input> <soap12:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap12:body use="literal"/> </wsdl:output>
      <wsdl:fault name="GetCertificateError"> <soap12:fault
name="GetCertificateError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
    <wsdl:operation name="GetComponent">
      <soap12:operation soapAction="http://mades.entsoe.eu/GetComponent"/>
      <wsdl:input> <soap12:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap12:body use="literal"/> </wsdl:output>
      <wsdl:fault name="GetComponentError"> <soap12:fault name="GetComponentError"
use="literal"/> </wsdl:fault>
    </wsdl:operation>
    <wsdl:operation name="SetComponentMversion">
      <soap12:operation soapAction="http://mades.entsoe.eu/SetComponentMversion"/>
      <wsdl:input> <soap:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap:body use="literal"/> </wsdl:output>
      <wsdl:fault name="SetComponentMversionError"> <soap12:fault
name="SetComponentMversionError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="MadesDirectoryService">
    <wsdl:port name="MadesDirectoryServiceSOAP11"
binding="mades:MadesDirectoryServiceSOAP11">
      <soap:address location="http://mades.entsoe.eu"/>
    </wsdl:port>
    <wsdl:port name="MadesDirectoryServiceSOAP12"
binding="mades:MadesDirectoryServiceSOAP12">
      <soap12:address location="http://mades.entsoe.eu"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

7.6.3.4 Node synchronization interface

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="MadesNodeSynchronizationService"
targetNamespace="http://mades.entsoe.eu/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:ecp="http://mades.entsoe.eu/"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/">

  <wsdl:types>
    <xsd:schema targetNamespace="http://mades.entsoe.eu/">
```

```
      <xsd:element name="GetAllDirectoryDataRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0" name="dversion" nillable="true"
type="xsd:int"/>
            <xsd:element minOccurs="0" name="serviceMversion" nillable="true"
type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="GetAllDirectoryDataResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="dversion" type="xsd:int"/>
            <xsd:element name="nodeCode" type="xsd:string"/>
            <xsd:element maxOccurs="unbounded" minOccurs="0" name="components"
nillable="true" type="mades:ComponentDescription"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="GetAllDirectoryDataError">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="errorCode" type="xsd:string"/>
            <xsd:element name="errorID" type="xsd:string"/>
            <xsd:element name="errorMessage" type="xsd:string"/>
            <xsd:element minOccurs="0" name="errorDetails" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="GetNodeMversionRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="mversion" type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="GetNodeMversionResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="mversion" type="xsd:int"/>
            <xsd:element name="nodeCode" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:element name="GetNodeMversionError">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="errorCode" type="xsd:string"/>
            <xsd:element name="errorID" type="xsd:string"/>
            <xsd:element name="errorMessage" type="xsd:string"/>
            <xsd:element minOccurs="0" name="errorDetails" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>

      <xsd:complexType name="ComponentDescription">
        <xsd:sequence>
          <xsd:element name="information" type="mades:ComponentInformation"/>
          <xsd:element maxOccurs="unbounded" minOccurs="0" name="certificates"
type="mades:ComponentCertificate"/>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="ComponentCertificate">
```

```
        <xsd:sequence>
          <xsd:element name="certificate" type="mades:Certificate"/>
          <xsd:element minOccurs="0" name="revoked" nillable="true"
type="xsd:boolean"/>
          <xsd:element name="type" type="mades:CertificateType"/>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="Certificate">
        <xsd:sequence>
          <xsd:element name="certificateID" type="xsd:string"/>
          <xsd:element name="certificate" type="xsd:base64Binary"/>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:simpleType name="CertificateType">
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="AUTHENTICATION"/>
          <xsd:enumeration value="ENCRYPTION"/>
          <xsd:enumeration value="SIGNING"/>
        </xsd:restriction>
      </xsd:simpleType>

      <xsd:complexType name="ComponentInformation">
        <xsd:sequence>
          <xsd:element name="code" type="xsd:string"/>
          <xsd:element name="type" type="mades:ComponentType"/>
          <xsd:element name="organization" type="xsd:string"/>
          <xsd:element name="person" type="xsd:string"/>
          <xsd:element name="email" type="xsd:string"/>
          <xsd:element name="phone" type="xsd:string"/>
          <xsd:element name="routing" type="mades:RoutingInformation"/>
          <xsd:element minOccurs="0" name="expiration" nillable="true"
type="xsd:long"/>
          <xsd:element minOccurs="0" name="codeMversion" nillable="true"
type="xsd:int"/>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="RoutingInformation">
        <xsd:sequence>
          <xsd:element name="node" type="xsd:string"/>
          <xsd:element name="primaryURL" type="xsd:string"/>
          <xsd:element minOccurs="0" name="secondaryURL" nillable="true"
type="xsd:string"/>
          <xsd:element minOccurs="0" name="nodeMversion" nillable="true"
type="xsd:int"/>
        </xsd:sequence>
      </xsd:complexType>

      <xsd:simpleType name="ComponentType">
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="NODE"/>
          <xsd:enumeration value="ENDPOINT"/>
        </xsd:restriction>
      </xsd:simpleType>

    </xsd:schema>
  </wsdl:types>

  <wsdl:message name="GetAllDirectoryDataResponse">
    <wsdl:part name="parameters" element="mades:GetAllDirectoryDataResponse"/>
  </wsdl:message>

  <wsdl:message name="GetAllDirectoryDataFault">
    <wsdl:part name="fault" element="mades:GetAllDirectoryDataError"/>
  </wsdl:message>

  <wsdl:message name="GetAllDirectoryDataRequest">
    <wsdl:part name="parameters" element="mades:GetAllDirectoryDataRequest"/>
```

```
  </wsdl:message>

  <wsdl:message name="GetNodeMversionResponse">
    <wsdl:part name="parameters" element="mades:GetNodeMversionResponse"/>
  </wsdl:message>

  <wsdl:message name="GetNodeMversionFault">
    <wsdl:part name="fault" element="mades:GetNodeMversionError"/>
  </wsdl:message>

  <wsdl:message name="GetNodeMversionRequest">
    <wsdl:part name="parameters" element="mades:GetNodeMversionRequest"/>
  </wsdl:message>

  <wsdl:portType name="MadesNodeSynchronizationService">
    <wsdl:operation name="GetAllDirectoryData">
      <wsdl:input message="mades:GetAllDirectoryDataRequest"/>
      <wsdl:output message="mades:GetAllDirectoryDataResponse"/>
      <wsdl:fault name="GetAllDirectoryDataError"
message="mades:GetAllDirectoryDataFault"/>
    </wsdl:operation>
    <wsdl:operation name="GetNodeMversion">
      <wsdl:input message="mades:GetNodeMversionRequest"/>
      <wsdl:output message="mades:GetNodeMversionResponse"/>
      <wsdl:fault name="GetNodeMversionError" message="mades:GetNodeMversionFault"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="MadesNodeSynchronizationServiceSOAP12"
type="mades:MadesNodeSynchronizationService">
    <soap12:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="GetAllDirectoryData">
      <soap12:operation soapAction="http://mades.entsoe.eu/GetAllDirectoryData"/>
      <wsdl:input> <soap12:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap12:body use="literal"/> </wsdl:output>
      <wsdl:fault name="GetAllDirectoryDataError"> <soap12:fault
name="GetAllDirectoryDataError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
    <wsdl:operation name="GetNodeMversion">
      <soap12:operation soapAction="http://mades.entsoe.eu/GetNodeMversion"/>
      <wsdl:input> <soap12:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap12:body use="literal"/> </wsdl:output>
      <wsdl:fault name="GetNodeMversionError"> <soap12:fault
name="GetNodeMversionError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:binding name="MadesNodeSynchronizationServiceSOAP11"
type="mades:MadesNodeSynchronizationService">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="GetAllDirectoryData">
      <soap:operation soapAction="http://mades.entsoe.eu/GetAllDirectoryData"/>
      <wsdl:input> <soap:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap:body use="literal"/> </wsdl:output>
      <wsdl:fault name="GetAllDirectoryDataError"> <soap:fault
name="GetAllDirectoryDataError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
    <wsdl:operation name="GetNodeMversion">
      <soap:operation soapAction="http://mades.entsoe.eu/GetNodeMversion"/>
      <wsdl:input> <soap:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap:body use="literal"/> </wsdl:output>
      <wsdl:fault name="GetNodeMversionError"> <soap:fault
name="GetNodeMversionError" use="literal"/> </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="MadesNodeSynchronizationService">
    <wsdl:port name="MadesNodeSynchronizationServiceSOAP12"
binding="mades:MadesNodeSynchronizationServiceSOAP12">
```

```
      <soap12:address location="http://mades.entsoe.eu"/>
    </wsdl:port>
    <wsdl:port name="MadesNodeSynchronizationServiceSOAP11"
binding="mades:MadesNodeSynchronizationServiceSOAP11">
      <soap:address location="http://mades.entsoe.eu"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

7.6.4 XML signature example

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-
20010315"/>
    <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha512"/>
    <Reference URI="">
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha512"/>
      <DigestValue>eVpInNsCIWzEjdrxxvongO2rnQ4=</DigestValue>
    </Reference>
  </SignedInfo>
      <SignatureValue>aD9HNiTmVxW+HnDOpSjzwDB+MypGTC7yb3/HUpAZKmEhRwQC0eBwYcZSRTqF8VdzmneH6abq2P+m
vHNXPC53i3mF58XDR5JFHHWLHq8B9HZm6/IYxcNy2cGW9yAVyQKe3uJXeV/95u9qMEwJhbOjvPIx
ZdbXqcCSorWqih7hdB86Nv2SIBfXMvWdIinwZfU/44RUptNyxQpP/Pw91Dd8YnMTNVwm2ax5oL1W
akIGKToS/yYid/Cgyb1xhGdfXEp30bqLusaLMYkbctpZ2WDn2w5I4mmOO78jndPUnaMT5gyFaonz
+K84xD+1/tZbTQ0adc9LE7XgAkpiiNjf2LW9tw==</SignatureValue>
  <KeyInfo>
     <KeyName>yyyyyyyyyyyyyyy</KeyName>
  </KeyInfo>
</Signature>
```

Where:

- DigestValue is the non encoded hash of the message.
- SignatureValue is the encoded hash of the message.
- KeyName is the ID of the signer component.

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

3, rue de Varembé
PO Box 131
CH-1211 Geneva 20
Switzerland

Tel: + 41 22 919 02 11
Fax: + 41 22 919 03 00
info@iec.ch
www.iec.ch