International Standard

ISO 4891

First edition
2024-11

Ships and marine technology — Interoperability of smart applications for ships

Navires et technologie maritime — Interopérabilité des applications intelligentes pour les navires

Reference number
ISO 4891:2024(en)

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of ISO document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

ISO draws attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO takes no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents. ISO shall not be held responsible for identifying any or all such patent rights.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/TC 8, *Ships and marine technology*.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

Introduction

In 2016, an exchange with ship managers and authorities was held to discuss why the digitalization of the market was behind that of other industries.

Due to new regulations and the desire to increase efficiency, the shipping industry requires ever more digital data and expertise. However, this can lead to high manual efforts and distractions.

In response to demands from stakeholders, a fast-growing digitalization process was initated. This digitalization has nevertheless been lagging in terms of the need for fast and reliant data collection and utilization.

In 2017, the exchange was widened to cover demands from of employees from ship to shore, to ensure that products and solutions are applicable also for crew members and linked maritime stakeholders. In particular, the following were addressed: data-collection, workflow support, automation and compatibility with other stakeholders, the IoT, ship equipment and other applications and standards.

The ideas and requirements that emerged from this exchange have been developed and summarized in this document in order to define a common base for the interoperability of digital devices onboard in collaboration with international experts.

In the process, a modular basis has been created to enable new applications on ships and promote the idea of preparing ship-related stakeholders for future issues, thus enabling them to work together in a secure and trustworthy manner.

In order to meet future requirements and to enable synergies between topics and stakeholders, this document aims to build on the existing technical basis and to add further mutually compatible modules or applications.

The prefix “4891” (this document's ISO number) is used for some terms throughout this document to differentiate those terms from similar terms of other documents or standards (e.g. 4891-component vs. component).

Ships and marine technology — Interoperability of smart applications for ships

1 Scope

This document provides operational and performance requirements for smart applications on board ships. It is applicable to documentation, process management, connection and data collection through human-machine interfaces, IoT technologies and related systems.

This document defines methods to implement smart network applications, which are open to participants who implement the requirements defined in this document.

This document also describes a smart logbook application that can be used as a supplement to ISO 21745, thus this document is subject to the same security requirements as in ISO 21745 (see Annex B).

This document defines three incremental levels of equipment-classes (see 7.1.1):

a) 4891-compliant equipment (as described in Clauses 5 to 7);

b) 4891.A-compliant equipment (as described in Clauses 5 to 7 and Annex A);

c) 4891.B-compliant equipment (as described in Clauses 5 to 7 and Annexes A and B).

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEC 61162-450:2018, *Maritime navigation and radiocommunication equipment and systems — Digital interfaces — Part 450: Multiple talkers and multiple listeners - Ethernet interconnection*

IEC 61162-460, *Maritime navigation and radiocommunication equipment and systems — Digital interfaces — Part 460: Multiple talkers and multiple listeners - Ethernet interconnection - Safety and security*

ISO 21745:2019, *Electronic record books for ships — Technical specifications and operational requirements*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

3.1
4891-component
functional part of the *smart application network* (3.26), such as the *4891-unit registry* (3.8), the *4891-service discovery* (3.6), the *4891-message broker* (3.5) or any *4891-unit* (3.7)

3.2
4891-smart gateway unit
optional *4891-unit* (3.7) that acts as central gateway and integration point between the *smart application network* (3.26) and *controlled equipment* (3.15)

3.3
4891-I/O unit
optional *4891-unit* (3.7) that acts as input, output and processing device implementing use case specific functionality

Note 1 to entry: I/0 units support manual input or automatic collection via sensors of any kind and may provide additional functions.

3.4
4891-message
structured piece of data that is exchanged between *4891-units* (3.7) and used in communication with *controlled equipment* (3.15)

3.5
4891-message broker
central *4891-component* (3.1) that can be used by *4891-units* (3.7) to publish and subscribe to *4891-messages* (3.4)

3.6
4891-service discovery
central 4891-component (3.13) that provides connectivity information about the other *central 4891-components* (3.13)

3.7
4891-unit
4891-smart gateway unit (3.2) or *4891-I/O unit* (3.3) that exchanges *4891-messages* (3.4) with other *4891-units* (3.7) or communicates with *controlled equipment* (3.15)

3.8
4891-unit registry
central 4891-component (3.13) that acts as certificate authority and participant lookup of *4891-units* (3.7)

3.9
administrator
admin
authorized person, who has the capability to setup and configure the *central 4891-components* (3.13) via administration interfaces (e.g. admin web interfaces)

3.10
API endpoint
single part of an API that is addressed via a URL for accessing data or triggering a functionality

3.11
asymmetric cryptography
digitally encrypting data and verifying signatures with *public key* (3.23) information and decrypting and signing data with *private key* (3.22) information

3.12
base URL
URL that is used as a prefix when constructing URLs to reach *API endpoints* (3.10)

3.13
central 4891-component
4891-component (3.1), that is not a *4891-I/O unit* (3.3)

EXAMPLE *4891-message broker* (3.5), *4891-service discovery* (3.6), *4891-unit registry* (3.8) or *4891-smart gateway unit* (3.2).

3.14
certificate
data structure containing a *public key* (3.23) and identity information (e.g. an identifier or name) that is digitally signed by another trusted party, binding the key to the identity

3.15
controlled equipment
security and safety related equipment, i.e. equipment according to IEC 61162-450/ IEC 61162-460

EXAMPLE The electronic record book.

3.16
data encryption
transformation of original data with the intention that the original data can only be transformed back by its designated recipient

3.17
instrumentation
running of software in a special mode that allows measuring of performance and tracing of code execution and information

3.18
key-pair
pair of related *public keys* (3.23) and *private keys* (3.22) used in *asymmetric cryptography* (3.11)

3.19
message relaying
receiving a *4891-message* (3.4) that is targeted to a different *4891-unit* (3.7) with the intention of forwarding that message to that unit in some way

3.20
mobile device
non-stationary data processor that can be moved around

EXAMPLE Smartphones and tablets.

3.21
PEM encoding
common format for serializing cryptographic components such as *private keys* (3.22), *public keys* (3.23), and *certificates* (3.14)

3.22
private key
secret part of a cryptographic *key-pair* (3.18) that should be kept a secret and is used for *signing data* (3.24) or decrypting data

3.23
public key
public part of a cryptographic *key-pair* (3.18) that can be shared and is used for validating signed data or *data encryption* (3.16)

3.24
signing data
attaching an additional signature data to an original data with the purpose that any modification of the original data is discoverable

3.25
smart application
function for mobile devices which makes it possible to confidentially collect data directly on board ships, evaluate it and control procedures

3.26
smart application network
SAppNet
sappnet
4891-network
set of *central 4891-components* (3.13) and all registered *4891-units* (3.7) that communicate via the exchange of *4891-messages* (3.4)

3.27
X.509 certificate
common format for storing and exchanging digital *certificates* (3.14)

Note 1 to entry: For further information on X.509, see RFC 5280.

4 Abbreviated terms

API	application programming interface
CA	certification authority
DSA	digital signature algorithm
ECDSA	elliptic curve digital signature algorithm
ELRB	electronic record book (see ISO 21745)
EUT	equipment under test
HTTP	hypertext transfer protocol (see RFC 2616)
IDE	integrated development environment
IoT	Internet of Things
JSON	JavaScript® object notation (see RFC 8259)
LAN	local area network
MQTT	message queuing telemetry transport (see ISO/IEC 20922:2016)
PEM	privacy-enhanced mail
PKI	public key infrastructure
RSA	RSA cryptosystem (Rivest–Shamir–Adleman)
SCRAM	salted challenge response authentication mechanism (see RFC 5802)
SAppNet	smart application network
SHA	secure hash algorithm
OOW	officer of the watch
UDP	user datagram protocol
URL	uniform resource locator
USB	universal serial bus
UTC	coordinated universal time

UUID universally unique identifier

5 Smart application network

5.1 Overview

A smart application network (see Figures 1 to 4), is an uncontrolled network focusing on uncritical interactions and contents. Each device may join the network at any time, and disconnect and reconnect as necessary. Communication between the network participants is done by passing messages in specified ways.

The network consists of the message exchange semantics between its various participants. The document defines the message structure as a common language to be used.

To ensure compatibility between different manufacturers, an implementation based on the interfaces described in Clause 6 shall be fulfilled.

If controlled equipment is connected to the smart application network via a 4891-smart gateway unit, then that controlled equipment shall be implemented in accordance with the requirements of Annex A. If that controlled equipment represents an ELRB, then that controlled equipment shall furthermore be implemented in accordance with Annex B (see Figures 3 to 4).

Figure 1 illustrates the positioning of this new network in relation to other existing onboard networks and standards. Figures 2 to 4 give examples of three different configurations and show how the smart application network connects with other existing networks and systems.

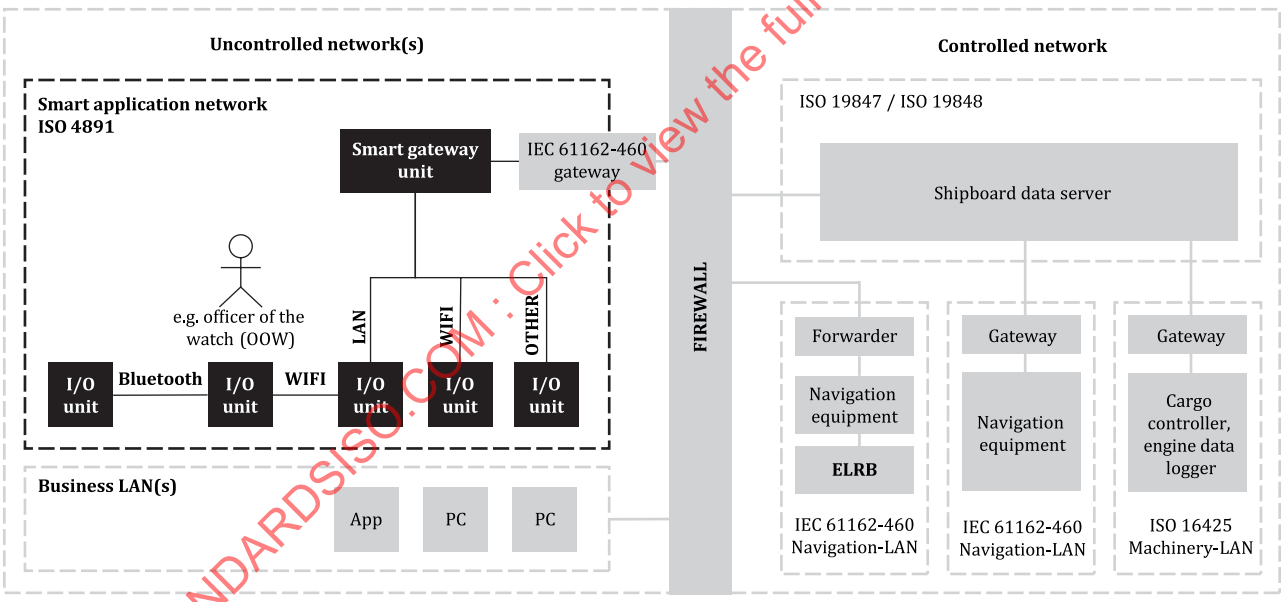
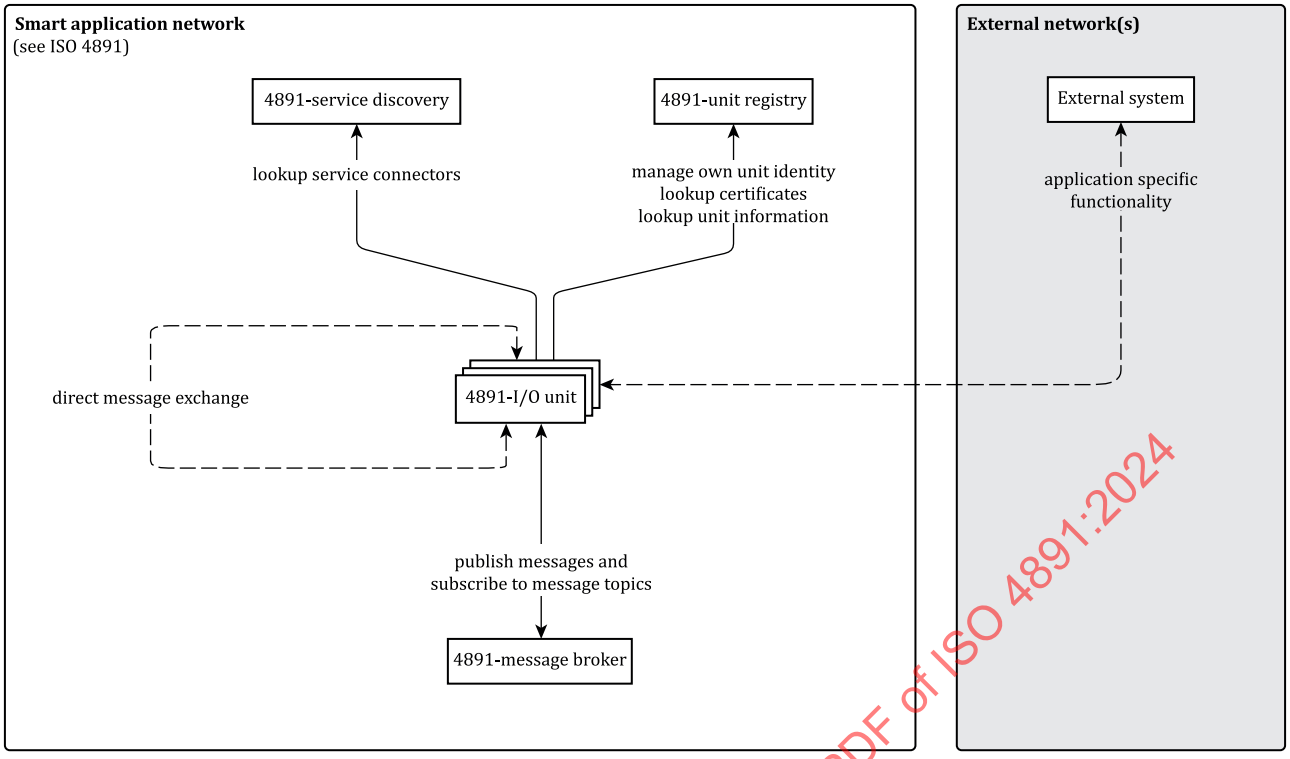


Figure 1 — Connectivity of 4891-network with other existing vessel networks



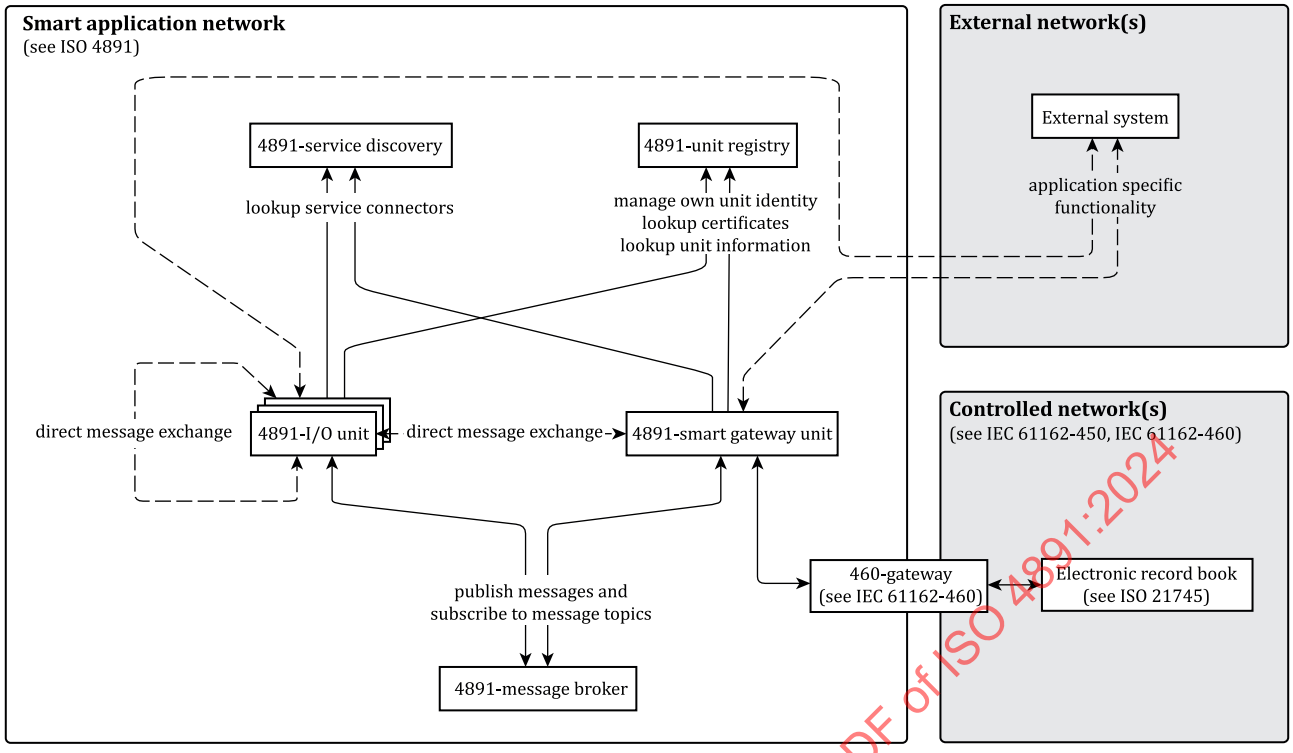
Key

→ data communication (e.g. via ethernet, wifi, bluetooth)

⇢ optional data communication

NOTE This configuration uses the equipment specified in this document, excluding the equipment specified in Annex A and Annex B.

Figure 2 — Configuration example — Basic smart applications setup



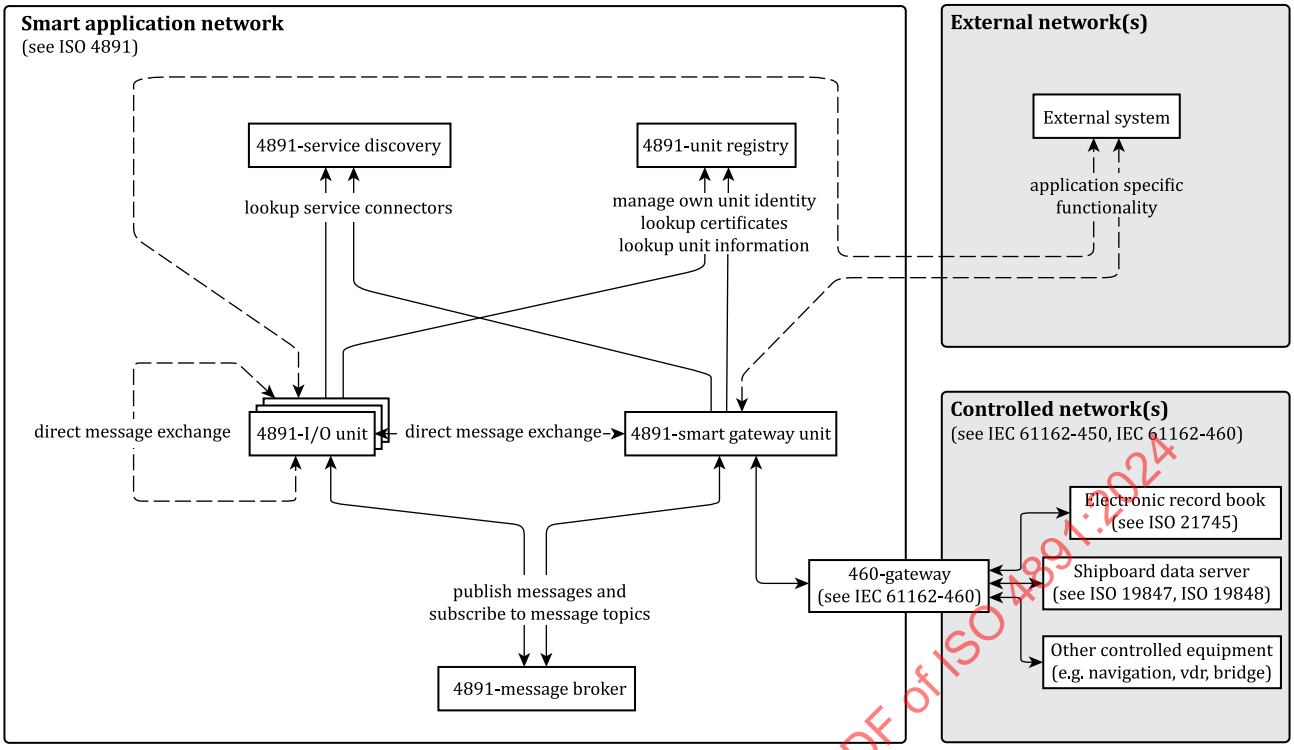
Key

→ data communication (e.g. via ethernet, wifi, bluetooth)

⇢ optional data communication

NOTE This configuration uses equipment specified in this document, including in Annex A and Annex B.

Figure 3 — Configuration example — Smart logbook as supplement to ELRB



Key

——→ data communication (e.g. via ethernet, wifi, bluetooth)

- - - - - - → optional data communication

NOTE This configuration uses equipment specified in this document, including in Annex A and Annex B.

Figure 4 — Configuration example — Smart logbook as supplement to ELRB with additional functions

5.2 4891-components

5.2.1 General

The smart application network shall be composed of network entities as listed in Table 1.

Table 1 — 4891-component quantities in smart application network

4891-component	Mandatory	Quantity
Message broker	Yes	1
Service discovery	Yes	1
Unit registry	Yes	1
Smart gateway unit	No	1
I/O unit	No	Unlimited

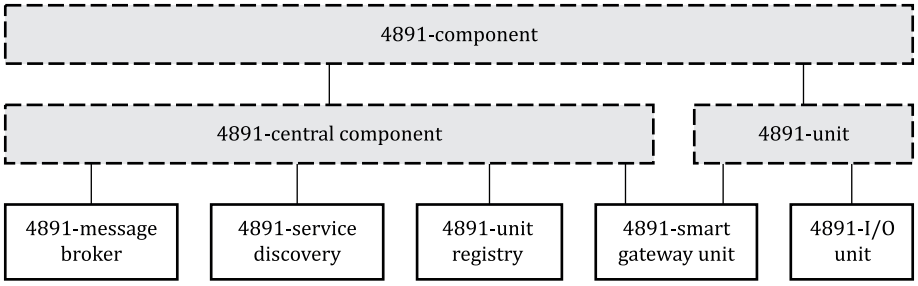


Figure 5 — Hierarchy of 4891-components

The message broker, service discovery, unit registry and smart gateway unit are grouped as central 4891-components (see Figure 5). Together with the I/O units, these are called 4891-components (see Figure 5).

The smart gateway unit shall be installed onboard of the vessel if connectivity between smart application network and controlled equipment is desired (see Annex A).

The 4891-network shall contain at most one 4891-smart gateway unit acting as a central gateway to controlled onboard networks (see Figures 3 and 4). Additionally, a conceptually unlimited number of 4891-I/O units may be added to the network, permanently or temporarily.

Manufacturers may enforce a limit to the maximum number of 4891-I/O units for technical reasons. Such a number shall be stated in the manufacturer's documentation.

5.2.2 4891-message broker

A central message broker shall be part of the smart application network.

The broker shall accept incoming 4891-messages from 4891-units, which are then sent to other interested 4891-units.

The broker shall allow 4891-units to subscribe to message topics. Those topics shall act as filters. The broker shall send the messages fulfilling the filter criteria to the subscribed 4891-units. At least the following message filters shall be supported:

— message type;

— message source unit;

— message destination unit;

— message destination kind: single destination versus broadcast.

The 4891-message broker can be unreliable when delivering a message. In such cases, 4891-units shall take suitable counter-measurements to enforce the reliability of the message (i.e. by resending and acknowledging messages).

The 4891-message broker may provide a buffering functionality to retry delivering messages, so as to improve delivery behaviour in an unstable connected network.

5.2.3 4891-service discovery

The 4891-service discovery shall provide connectivity information (“connectors”) about the other central 4891-components. These connectors include:

— 4891-message broker;

— 4891-unit registry;

— 4891-smart gateway unit.

Those connectors shall be queryable by all 4891-components.

The service discovery shall have a functionality to configure those connectors by authorized persons. How such configuration can be accomplished shall be described in the manufacturer's documentation.

5.2.4 4891-unit registry

The 4891-unit registry shall provide the functionality for equipment to register itself as a 4891-unit when logically joining or leaving the smart application network. The list of registered 4891-units shall be queryable from the registry by all 4891-components.

The 4891-unit registry shall also act as the certificate authority for trusted message exchange in the smart application network. The 4891-unit registry shall provide the registering units with data that can be used to encrypt, decrypt and sign messages. Furthermore, the registry shall provide the functionality for 4891-units to lookup information, allowing them to encrypt, decrypt and sign messages as needed (see 5.7), i.e. by using PKI, issuing signed unit certificates, and allowing lookup of public keys of units.

5.2.5 4891-units

5.2.5.1 General

A 4891-unit is part of a manufacturer-specific application that is implemented on top of the smart application network. For that reason, one or more units perform logic and exchange information.

4891-units shall provide functionality to create and process 4891-messages as a mechanism to exchange information with other 4891-units.

4891-units may exchange 4891-messages directly with other 4891-units (i.e. direct messaging, see 5.5), indirectly via other 4891-units (i.e. message relaying, see 5.6) or via the central 4891-message broker, depending on capabilities and purpose of units.

Two types of 4891-units are distinguished by their intended purpose:

— the smart gateway unit is intended for integration with other onboard equipment in controlled networks;

— the I/O unit is intended for data collection, data processing and data interaction by users.

5.2.5.2 4891-smart gateway unit

The 4891-smart gateway unit is an optional central component of the smart application network. It is the integration point with other potentially controlled networks. In that regard, the 4891-smart gateway unit acts as a gateway and firewall between the controlled equipment of those networks and the 4891-components.

The 4891-smart gateway unit regulates data access between 4891-I/O units and systems in controlled networks depending on system specific requirements.

If a 4891-smart gateway unit is part of the 4891-network, it shall be permanently connected to the 4891-message broker, subscribing at least to 4891-messages targeted at the smart gateway unit.

If existing, the 4891-smart gateway unit shall be installed onboard of the vessel.

If the 4891-smart gateway is connected to navigational or controlled equipment, then it shall be done in accordance with Annex A.

5.2.5.3 4891-I/O units

The 4891-I/O units represent the smart devices in the smart application network.

The boundary of an I/O unit is not strictly a single piece of hardware. Any logically distinct computation unit that conforms to the requirements for I/O units is considered an I/O unit. This does potentially include mobile devices, stationary hardware, or a piece of software. A single piece of hardware may contain multiple distinct I/O units, each being distinctively addressable via 4891-messages.

4891-I/O units may either be installed permanently or may join and leave the logical network dynamically depending on the use cases that they are intended to accomplish. As long as an I/O unit is not losing its cryptographic proof of identity (see 5.7), it continues being considered a part of the logical network even when leaving and joining the physical network layers. This allows for "eventually connected" logical network topologies of units in the smart application network.

The potentially mobile nature of I/O units combined with the optional message relaying functionality allows for forming an eventually connected peer to peer topology between the units.

5.3 4891-messages

5.3.1 General

5.3 defines the general logical structure of messages that are used for data exchange between 4891-units.

Specific message types can be defined for proprietary use cases. Some basic message types and their semantics are defined in Clause 6, Annex A and Annex B.

5.3.2 Message structure

All 4891-messages are structured in the following way (see Figure 6):

— each message is composed of at least one data part, or none;

— each message has headers for meta-information about the message;

— each data part has a payload encoded in one of the supported ways;

— each data part has headers for meta-information about the data part.

5.3.3 to 5.3.6 define the different structural elements of a message in more detail.

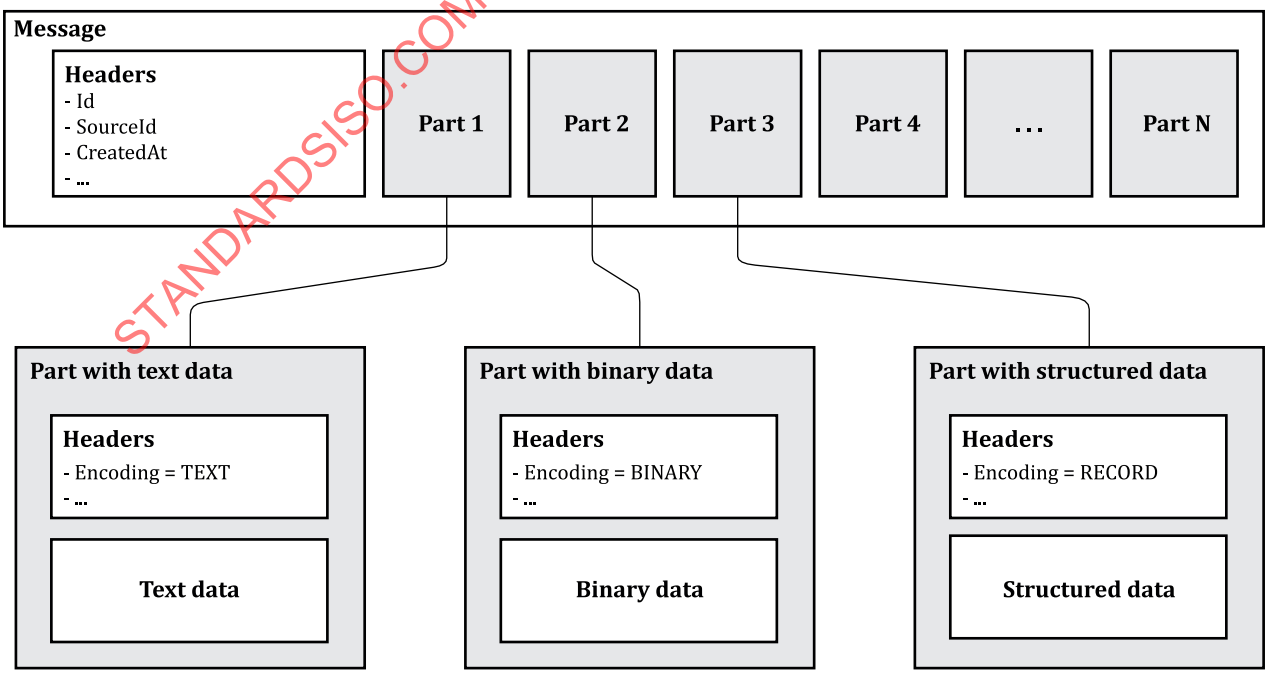


Figure 6 — Logical structure of a 4891-message

5.3.3 Header values

5.3.3.1 General

The message and its data parts have headers for transporting meta-information. Each header is composed of a header name and a header value.

Header names shall be composed of these characters only:

— `a` to `z` (lowercase letters);

— `A` to `Z` (uppercase letters);

— `0` to `9` (numeric digits).

Header names are case-insensitive, e.g. units shall consider header names `CreatedAt` and `createdat` to be equal and synonyms for the same header.

Messages and data parts can have varying numbers of headers, depending on the kind of messages being transmitted. Tables 2 and 3 define the basic headers that shall be used when dealing with messages.

There are two types of headers, as shown in Table 2.

Table 2 — Types of 4891-message headers

Header type	Description	Examples
Standard header	All headers not beginning with an `x` character.	`Id`, `SrcUnitId`, `CreatedAt`
Non-standard header	All headers beginning with a `x` character.	`XCustomMetaInfo`, `xsomething`

Units shall accept incoming messages that contain unknown standard headers (headers with names not specified in this document) but shall ignore them when processing the message.

Units shall not emit messages with standard headers that are not defined in this document. The only exception are messages that they received for message relaying. In case of message relaying, unknown standard headers shall not be removed from messages before being forwarded to another unit.

Non-standard headers can be used to transmit proprietary data. Implementations may transmit any data in such headers for any purpose. Units that interpret non-standard headers shall be aware that different implementations can use the same header names for different purposes. When using such headers, a manufacturer shall pick any suitable name for the field and prefix it with an `X` and a short string representation of the manufacturer's name as part of the fields name (e.g. `XManufacturernameField`). Different manufacturers may use the same non-standard headers for different purposes.

5.3.3.2 Standard message headers

Table 3 lists the base standard message headers. Additional standard message headers are defined in Annex A.

Table 3 — Standard 4891-message headers

Header name	Value type	Required	Description
`Id`	`<uuid>`	yes	An identifier for the message, unique for the equipment that created the message
`Type`	`<enum>`	yes	The message type indicates the semantic of the message and imposes additional constraints on the message's content, e.g. number of data parts, headers, record structure, ... (see 5.3.5)
`SrcUnitId`	`<uuid>`	yes	The source unit of the message, i.e. the unit that originally emitted the message
`DestUnitId`	`<uuid>`	no	The destination unit of the message, i.e. the unit that should ultimately receive the message for processing
`CreatedAt`	`<timestamp>`	yes	Date and time of the message's creation by the source unit
`ExpiresAt`	`<timestamp>`	no	Timestamp defining the message's expiration date and time
`ReferenceId`	`<uuid>`	no	Indicates that this message references another message, e.g. for request-response message chaining.

The `ReferenceId` header shall be set on messages created as a response to another message. The value of that header shall be the `Id` value from the message that is responded to.

5.3.3.3 Standard data part headers

Table 4 defines the standard data part headers.

Table 4 — Standard data part headers of 4891-messages

Header name	Value type	Required	Description
`Encoding`	`<enum>`	yes	This header defines how the part's data are encoded. The allowed values are defined in Table 5. See 5.3.4.1 for more details.

5.3.4 Data part encoding

5.3.4.1 General

The payload of a data part is used to transport the actual data of the message. The data part header `Encoding` is used to specify the encoding of the data part, as shown in Table 5.

Table 5 — Values for data part header `Encoding`

Header value	Description
`TEXT`	The payload is a simple UTF-8 encoded text sting.
`BINARY`	The payload is encoded in binary. This encoding is suitable to transport binary files.
`RECORD`	The payload is a structured record value, see 5.3.4.2.

5.3.4.2 Structured record payload

A `RECORD`-encoded payload is a represented by a key-value dictionary structure. Each value in that structure can be one of the following:

— text value;

— integer number value;

— list of values;

— dictionary of values.

The record's structure and nesting depth (e.g. when using lists and dictionaries as values) depends on the message's type. Record definitions shall be part of any message type definition where the message type declares usage of any `RECORD`-encoded message parts.

5.3.5 Message type

Each 4891-message shall have a concrete type, declared in the message's `Type` header.

The message type binds semantical meaning and expectations about the message's structure and content to the message.

Message types shall be composed only of these characters:

— `a` to `z` (lowercase letters);

— `A` to `Z` (uppercase letters);

— `0` to `9` (numeric digits);

— `.` (point).

Message types shall be handled case-insensitive, e.g. units shall consider message type strings like `SOME.MessageType` and `some.messagetype` to be equal and synonyms for the same message type.

Like header values, the message type's value space is divided into two groups:

a) type not beginning with the `X` character is a "standard message type";

b) type beginning with the `X` character is a "non-standard message type"

Units shall not emit messages with standard message types that are not defined in this document. The only exception are messages that they received for message relaying.

Non-standard message types shall be described in the manufacturer's documentation, containing the following information:

— headers which are expected;

— number of data parts;

— meaning and encoding of data parts;

— record structures if any data parts are `RECORD`-encoded;

— situations when to send messages of this type;

— actions after receiving messages of this type;

— senders and receivers for messages of this type.

Non-standard message types can be used to transmit proprietary data. Implementations are allowed to transmit any data in such messages for any purpose. Units that interpret messages with non-standard message types shall be aware that different implementations can use the same types for different purposes. When using such types, a manufacturer shall pick any suitable name for the type and prefix it with an `X` and a short string representation of the manufacturer's name as part of the type name (e.g. `XManufacturernameSomeType`). Different manufacturers may use the same non-standard message types for different purposes.

5.3.6 Standard message types

This document defines a set of standard message types together with their payload structure and required semantics. Table 6 lists those message types and refers to where to find details about them.

Additional standard message types may be defined in other standards relating to this document.

Manufacturers are free to define additional non-standard message types based on their use case requirements.

Table 6 — Index of standard 4891-message types in this document

Standard message type	Related clause
`base.signed`	6.10.5
`base.encrypted`	6.10.6
`base.error`	6.11.2
`smgw.gateways.query`	A.10.2
`smgw.gateways.queryresp`	A.10.2
`smgw.users.auth`	A.10.3
`smgw.users.authresp`	A.10.3
`smlog.records.query`	B.5.2
`smlog.records.queryresp`	B.5.2
`smlog.records.export`	B.5.3
`smlog.records.exportresp`	B.5.3
`smlog.records.write`	B.5.4
`smlog.records.delete`	B.5.5
`smlog.records.verify`	B.5.6
`smlog.records.verifyrange`	B.5.6

5.4 Handling of outdated messages

4891-messages may have an expiry timestamp as part of their meta-information (see 5.3.3.2). That value transports an instant in time, after which the message shall be considered as being “outdated”.

4891-units receiving outdated messages that are intended for a different 4891-unit (see 5.6) shall discard them and not forward them any further.

5.5 Direct messaging

4891-units may provide the functionality to receive messages directly addressed to them by other 4891-units. This option allows direct ad hoc communication between two units instead of using the central 4891-message broker.

Manufacturers shall decide if their 4891-unit provides this functionality or not. If provided, the 4891-units shall at least support the exchange of 4891-messages via the direct messaging functionality specified in 6.9. Additional ways of communication between 4891-units may be implemented.

5.6 Message relaying

4891-units may accept 4891-messages intended for other 4891-units with the purpose of forwarding those messages.

When such a relaying functionality is provided and used in conjunction with message encryption or signing, it shall not be possible for the relaying units to decrypt the messages or modify their content without breaking signatures.

5.7 Trust and encryption

Whenever the trustworthiness of handled data is important, it shall be digitally signed by the emitter of the data. This shall be done in a way that the message destination is able to validate the signature.

Whenever data are considered to contain private information that must be protected against eavesdropping, this data shall be encrypted. This shall be done in a way that only the desired recipient of the message is able to decrypt it.

The necessity of data trustworthiness and data encryption depends on the actual use cases to be implemented in the smart application network.

6 Compatibility implementation

6.1 General

Clause 6 describes the technologies, protocols, and interfaces to be used by implementations of this document to reach interoperability and compatibility between 4891-components from different manufacturers.

All requirements described in Clauses 5 and 6 shall be met for an implementation to be considered as conforming to a smart application network. Additional ways of communication between 4891-components may be implemented.

Manufacturers may choose to implement the full set of 4891-components (i.e. message broker, service discovery, unit registry, smart gateway unit, I/O unit) or subsets of it. The manufacturer's documentation shall state which 4891-components are implemented.

Clause 6 defines concrete interfaces and processes for requirements of the different 4891-components described in Clause 5. Additionally, a JavaScript®[1] object notation (JSON)-encoding scheme for 4891-messages and the process for signing and encrypting 4891-messages are specified.

An extension building on these requirements is defined in Annex A (i.e. integrating with controlled equipment) and in Annex B (i.e. specific use case “smart logbook”).

6.2 JSON-encoding for value types

6.2.1 General

6.2 provides a JSON-encoding schema for all value types used in this document. This encoding shall be used for all data payloads exchanged in the interfaces described in this document, e.g. for HTTP-API requests, responses, and query parameters and when transferring 4891-messages via MQTT-messages.

6.2.2 Common types

Table 7 lists the JSON-encoding for common types used in this document.

1) JavaScript is a registered trademark of Oracle Corporation. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC.

Table 7 — JSON-encoding for common data types

Type name	JSON type	Extended description
`<null>`	`null`	The null-value, denoting non-existence if a value.
`<integer>`	`number`	A 64-bit signed integer. The expected practical value range depends on the specific use case.
`<boolean>`	`boolean`	A boolean value of `true` or `false`.
`<string>`	`string`	A string value that can contain any UTF-8 encoded value.
`<enum>`	`string`	A string value that has a limited set of allowed values. This set may be closed or extendible, i.e. so that additional values can be added to this set in specific use cases. The set and its nature depend on the situation where this type is used.
`<uuid>`	`string`	A UUID encoded as string. EXAMPLE 1 — `"00000000-0000-0000-0000-000000000000"` — `"1b595396-7ccd-4cc1-b966-4851ed108515"`
`<binary>`	`string`	Binary data encoded as a Base64-encoded string (see RFC 4648). Example: `"aGVyZSBzb21lIGRhdGE = "` (decoded as string: `"here some data"`)
`<timestamp>`	`string`	A date-time value with seconds-accuracy and time zone, encoded as string and formatted as follows (see ISO 8601 series): `"<YYYY>-<MM>-<DD>T<hh>:<mm>:<ss><tz>"` EXAMPLE 2 — `"2021-12-30T12:45:56+02:00"` (UTC+2 time zone) — `"2021-12-30T12:45:56Z"` (UTC time zone)
`<url>`	`string`	A full-qualified URL-string, i.e. containing protocol, host and potentially a port, path, and query parameters. EXAMPLE 3 — `"http://192.168.10.17"` — `"http://192.168.10.17:8080/some%20path/calc?x=42"`
`<base-url>`	`string`	Like `<url>` but without any query-parameters. A value of this type is used as prefix to construct full URL strings by appending additional path segments and potentially query parameters to it. Values of this type shall never end with a trailing `/`-character. EXAMPLE 4 — `"http://192.168.10.17"` — `"http://192.168.10.17:8080/some%20path"`
`<certificate-pem>`	`string`	A PEM-encoded string containing an X.509 certificate.
`<publickey-pem>`	`string`	A PEM-encoded string containing a public key.
`<list of x>`	`array`	This type describes a list of type `<x>`, which is encoded as an array of the encoded values for that type.

6.2.3 Dictionary type

Table 8 lists the JSON-encoding for the `dictionary` type used in this document.

Table 8 — JSON-encoding for type “<dictionary>”

<table>
<tr><th>Type name</th><th>JSON type</th><th colspan="2">Extended description</th></tr>
<tr><td><dictionary></td><td>object</td><td colspan="2">A dictionary is a recursive type. The dictionary's fields are encoded as <string> in the object's keys. The dictionary's field values are encoded as <dict-field-value> in the object's values.</td></tr>
<tr><td rowspan="8"><dictionary-field-value></td><td rowspan="8">mixed</td><td colspan="2">Depending on the dictionary field's type this type maps to one of these types:</td></tr>
<tr><td>Field Type</td><td>Mapped Type</td></tr>
<tr><td>null value</td><td><null></td></tr>
<tr><td>string value</td><td><string></td></tr>
<tr><td>integer value</td><td><integer></td></tr>
<tr><td>boolean value</td><td><boolean></td></tr>
<tr><td>list value</td><td><list of dictionary-field-value></td></tr>
<tr><td>dictionary value</td><td><dictionary></td></tr>
</table>

EXAMPLE 1

JSON-encoding of an empty dictionary:

```
{}
```

EXAMPLE 2

JSON-encoding of a dictionary with one string-entry:

```
{
  "field":"value"
}
```

EXAMPLE 3

JSON-encoding of a dictionary with multiple entries of mixed types:

```
{
  "a": 123,
  "b": true,
  "c": "some string",
  "d": ["x", "y"],
  "e": {"z": null}
}
```

6.2.4 Message type

Table 9 lists the JSON-encoding for the value type <message> used in this document.

Table 9 — JSON-encoding for type “<message>”

<table>
<tr><th>Type Name</th><th>JSON Type</th><th>Extended Description</th></tr>
<tr><td><message></td><td>object</td><td>{
“header”: <message-header>
“parts”: <list of message-part>
}</td></tr>
<tr><td><message-header></td><td>object</td><td>Header values are encoded as a simple key-value maps with header names and their values being strings.</td></tr>
<tr><td><message-part></td><td>object</td><td>{
“header”: <message-header>
“data”: <message-part-data>
}</td></tr>
<tr><td><message-part-data></td><td>mixed</td><td>Depending on the part's encoding this type maps to one of these types:
<table><tr><th>Part encoding</th><th>Mapped type</th></tr><tr><td>TEXT</td><td>the string data as <string></td></tr><tr><td>BINARY</td><td>the binary data as <binary-string></td></tr><tr><td>RECORD</td><td>the record data as <dictionary></td></tr></table></td></tr>
</table>

EXAMPLE 1

JSON-encoding of an example message without any data parts:

```
{
  "header": {
    "Id": "b40ce24c-1d0b-42b8-9e25-08caf246f983",
    "Type": "some.message.type",
    "SrcUnitId": "ab8c336a-77da-439f-918f-daf7c238f269",
    "DestUnitId": "58675eb7-a9ea-4b7f-b80b-b0ebb8209016",
    "CreatedAt": "2021-03-03T12:45:56Z"
  },
  "parts": []
}
```

EXAMPLE 2

JSON-encoding of an example message with additional headers and three data parts:

```
{
  "header": {
    "Id": "81e2ecb2-f4d7-4b55-a6db-7ae034591924",
    "Type": "another.message.type",
    "SrcUnitId": "ab8c336a-77da-439f-918f-daf7c238f269",
    "DestUnitId": "58675eb7-a9ea-4b7f-b80b-b0ebb8209016",
    "CreatedAt": "2021-03-03T12:45:56Z",
    "ExpiresAt": "2021-03-05T12:45:56Z",
    "XSomething": "some non-standard header"
  },
  "parts": [
    {
      "header": {"Encoding": "TEXT"},
      "data": "I am some string payload."
    },
    {
      "header": {"Encoding": "BINARY"},
      "data": "SSBhbSBzb21lIGJpbmFyeSBjb250ZW50Lg=="
    },
    {
      "header": {"Encoding": "RECORD"},
      "data": {
        "the": ["record", "structure"],
```

```
          "as": "json",
          "with": {"some": "nesting", "and-number": 123}
        }
      }
    ]
  }
```

EXAMPLE 3

JSON-encoded message of type `base.signed` wrapping the two messages from Examples 2 and 3:

```
  {
    "header": {
      "Id": "56f42af1-c01e-4288-85f1-164fd35c6cae",
      "Type": "base.signed",
      "SrcUnitId": "ab8c336a-77da-439f-918f-daf7c238f269",
      "DestUnitId": "58675eb7-a9ea-4b7f-b80b-b0ebb8209016",
      "CreatedAt": "2021-03-03T12:45:56Z"
    },
    "parts": [
      {
        "header": {"Encoding": "TEXT"},
        "data": "[{\"header\":{\"Id\":\"81e2ecb2-f4d7-4b55-a6db-
7ae034591924\",\"Type\":\"some.message.type\",\"SrcUnitId\":\"ab8c336a-77da-439f-918f-daf7c2
38f269\",\"DestUnitId\":\"58675eb7-a9ea-4b7f-b80b-b0ebb8209016\",\"CreatedAt\":\"2021-03-03T-
12:45:56Z\"},\"parts\":[]},{\"header\":{\"Id\":\"81e2ecb2-f4d7-4b55-a6db-7ae034591924\",\"Typ
e\":\"another.message.type\",\"SrcUnitId\":\"ab8c336a-77da-439f-918f-daf7c238f269\",
\"DestUnitId\":\"58675eb7-a9ea-4b7f-b80b-b0ebb8209016\",\"CreatedAt\":\"2021-03-03T-
12:45:56Z\",\"ExpiresAt\":\"2021-03-05T12:45:56Z\",\"XSomething\":\"some non-standard header\"
},\"parts\":[{\"header\":{\"Encoding\":\"TEXT\"},\"data\":\"I am some string payload.\"},{\"he
ader\":{\"Encoding\":\"BINARY\"},\"data\":\"SSBhbSBzb211IGJpbmFyeSBjb250ZW50Lg==\"},{\"header\
":{\"Encoding\":\"RECORD\"},\"data\":{\"the\":[\"record\",\"structure\"],\"as\":\"json\",\"wit
h\":{\"some\":\"nesting\",\"and-number\":123}}}]}]"
      },
      {
        "header": {"Encoding": "BINARY"},
        "data": "… Base64-encoded signature data …"
      },
      {
        "header": {"Encoding": "TEXT"},
        "data": "… PEM-encoded X.509 certificate used for signing …"
      }
    ]
  }
```

EXAMPLE 4

JSON-encoded message of type `base.encrypted`:

```
  {
    "header": {
      "Id": "bd3ce1c6-3db7-4283-bcd8-120352594d5e",
      "Type": "base.encrypted",
      "SrcUnitId": "ab8c336a-77da-439f-918f-daf7c238f269",
      "DestUnitId": "58675eb7-a9ea-4b7f-b80b-b0ebb8209016",
      "CreatedAt": "2021-03-03T12:45:56Z"
    },
    "parts": [
      {
        "header": {"Encoding": "BINARY"},
        "data": "… Base64-encoded encrypted data …"
      }
    ]
  }
```

6.3 HTTP-APIs

6.3.1 General

All HTTP-APIs specified in this document shall use protocol version `1.1`, i.e. HTTP-servers and HTTP-clients shall be compatible with `HTTP/1.1` (see RFC 2616).

6.3.2 HTTP-requests

The following requirements are applicable to all 4891-components utilizing an HTTP-client to interact with HTTP-APIs specified in this document:

— Requests shall have HTTP-header "`Accepts: application/json`".

— Requests with content body shall have HTTP-header "`Content-Type: application/json`".

— Requests with content body shall have JSON-encoded content.

6.3.3 HTTP-request query parameters

For HTTP-request query parameters, the regular type encodings described in 6.2 shall be applied, with these differences:

— String values shall not be wrapped in quotes.

— Structured types or lists (i.e. objects and arrays in JSON) should be avoided for the use in HTTP-query parameters. If an endpoint uses such types nonetheless, the values of those types shall be encoded as JSON-strings.

— If the values are put into an URL string, those values shall be properly URL-encoded, escaping characters with special semantics in URLs (e.g. `?`, `=` , etc).

EXAMPLE 1

Number value:

```
1234
```

Query-encoding:

```
1234
```

Used as value on an URL parameter:

```
http://127.0.0.1?someParam=1234
```

EXAMPLE 2

String value:

```
1234
```

Query-encoding:

```
1234
```

Used as value on an URL parameter:

```
http://127.0.0.1?someParam=1234
```

EXAMPLE 3

String value containing special characters:

```
=/% string with special characters %\=
```

Query-encoding:

```
%3D%2F%25%20string%20with%20special%20characters%20%25%5C%3D
```

Used as value on an URL parameter:

```
http://127.0.0.1?someParam=%3D%2F%25%20string%20with%20special%20characters%20%25%5C%3D
```

EXAMPLE 4

String value containing JSON:

```
{"key":"value"}
```

Query-encoding:

```
%7B%22key%22%3A%22value%22%7D
```

Used as value on an URL parameter:

```
http://127.0.0.1?someParam=%7B%22key%22%3A%22value%22%7D
```

EXAMPLE 5

String value containing Base64-encoded binary (including padding):

```
aGVyZSBzb21lIGRhdGE=
```

Query-encoding:

```
aGVyZSBzb21lIGRhdGE%3D
```

Used as value on an URL parameter:

```
http://127.0.0.1?someParam=aGVyZSBzb21lIGRhdGE%3D
```


6.3.4 HTTP-responses

The following requirements are applicable to all 4891-components utilizing an HTTP-server for HTTP-APIs specified in this document:

— Responses with content body shall have HTTP-header "`Content-Type: application/json`".

— Responses with content body shall have JSON-encoded content.

The following requirements are applicable to responses for successful requests:

— Responses with content body shall have HTTP-status `200`.

— Responses without content body shall have HTTP-status `204`.

The following requirements are applicable to responses for failed requests:

— Responses shall have HTTP-status from the families `4XX`, `5XX` or `9XX`.

— Responses shall have a content body as described in HTTP-error responses.

6.3.5 HTTP-error responses

The following requirements are applicable to all 4891-components utilizing an HTTP-server or -client for HTTP-APIs specified in this document:

— Servers shall produce error responses for at least the situations listed in Table 11, additional error codes and possibilities are specified as part of specific API endpoint specifications.

— Clients shall detect error responses (i.e. by HTTP-status) and handle accordingly.

— Error responses shall have a content body containing the fields in Table 10.

Table 10 — HTTP-error response fields

Field	Value type	Required	Description
`code`	`<enum>`	yes	A technical short error code string. The expected values for this field shall be specified for operations that can fail. A list of common code values can be found in Table 11.
`details`	`<string>`	no	Optional details describing the error, describing what went wrong to help debugging. If text is present, English language shall be used for this field.

Table 11 lists error codes and HTTP-status values that shall be used in combination for typical error situations. Additional error codes may be added by implementations.

Table 11 — Common HTTP-error code values

Code value	HTTP-status	Description
`INVALID`	`400`	The request was invalid and has been rejected by the server.
`FORBIDDEN`	`401`	The endpoint requires proper authentication, but authentication data was either missing or invalid.
`NOT_FOUND`	`404`	The path was invalid or the resource behind that path has not been found.
`UNEXPECTED`	`500`	Some unexpected error has occurred while processing the request. This is the most generic fallback error code. Servers shall only respond with this error code if no other more descriptive error code is available for the error situation.

EXAMPLE

```
{
  "code": "INVALID",
  "details": "Unable to parse field \"id\" as UUID."
}
```

6.3.6 4891-unit authentication

6.3.6.1 General

Clause 6.3.6 is applicable if the HTTP-API provides endpoints that require a proper authentication of the 4891-unit using them. Those endpoints are called protected endpoints in Clauses 6 and 7 and Annexes A and B.

6.3.6.2 Endpoint RequestAuthToken

The HTTP-API shall provide an additional endpoint for requesting a one-time-usable auth-token.

The HTTP-server shall generate an auth-token for each request received on this endpoint and persist it.

Auth-tokens shall automatically expire 60 s after their creation. Implementations shall either flag such tokens or delete them.

Table 12 specifies the signature of this endpoint.

Table 12 — HTTP-signature of endpoint "RequestAuthToken"

Method	`POST`	
Path	`<ApiBaseUrl>/auth-token`	
Request	**Content**	`{}`

Response	**Content**	`{`
		`  "authToken": <string>`
		`}`

6.3.6.3 Protecting endpoints

The HTTP-server shall use this structure for request content on protected endpoints:

```
{
  "data": <string>
  "signature": <binary>
}
```

The `signature` field shall contain the signature of the `data` field, created with the sending unit's private key. The `data` field shall contain a JSON-encoded string of the following structure:

```
{
  "unitId": <uuid>
  "authToken": <string>
  "requestContent": <varying>
}
```

The `unitId` field shall contain the identifier of the unit authenticating against the protected endpoint, that is, the unit sending the request and signing the request. The `authToken` field shall contain an auth-token that had been created by the HTTP-server previously.

The `requestContent` field shall contain the actual content data for the protected endpoint. The structure of that field shall be specified as part of the protected endpoint's specification.

The HTTP-server shall perform the following steps when receiving a request on a protected endpoint:

a) Lookup the unit certificate for the unit identifier provided in the request.

b) Validate that the signature in the request relates to the unit certificate.

c) Validate that the signature is valid for the data field of the request for that unit certificate.

d) Check that the provided auth-token has not been used and has not expired yet.

If all these steps succeed, then the HTTP-server shall mark the provided auth-token as being used, e.g. by flagging or deleting it. Then the actual logic of the protected endpoint shall be performed as documented in the endpoint's specification. In that logic, the unit with the provided identifier shall be considered as being authenticated until the request has been fully handled.

If any of these steps fail, then the HTTP-server shall respond with an error-response with code `FORBIDDEN`.

6.3.6.4 Using protected endpoints

When interacting with a protected endpoint, the requesting HTTP-client shall perform these steps:

a) Request an auth-token each time before a protected endpoint will be called (see 6.3.6.3).

b) The received auth-token shall be placed in the request body content for the protected endpoint.

c) The protected endpoint's specific payload content, the received auth-token and a unit's id shall be encoded into a JSON-string and signed by the private key of the unit matching the provided unit id.

EXAMPLE

The following variable values are used in the example:

```
<ApiBaseUrl> = "http://192.168.3.17/the-api"
<UnitId>  = "2b985ef2-e305-4fae-ad5f-20630c9109a0"
```

Step 1

A new new auth-token is requested from the endpoint:

```
POST http://192.168.3.17/the-api/auth-token
```

Content sent in the request:

```
{}
```

Content received in the response:

```
{
  "authToken": "d7cbb91f6367ee5a73c25999d91c1"
}
```

Step 2

The data variable is built and encoded for the actual request.

The request-content for the protected endpoint:

```
requestContent = {"some":"value"}
```

The structured data value:

```
dataValue = {
  "unitId": "2b985ef2-e305-4fae-ad5f-20630c9109a0",
  "requestContent": {"some":"value"},
  "authToken": "d7cbb91f6367ee5a73c25999d91c1"
}
```

Encoded as JSON-string:

```
data = encodeJsonString(dataValue)
data = "{\"unitId\":\"2b985ef2-e305-4fae-ad5f-20630c9109a0\",\"requestContent\":{\"some\":\"
value\"},\"authToken\":\"d7cbb91f6367ee5a73c25999d91c1\"}"
```

Step 3

The `signature` variable is computed from the data string from step 2 and the sending unit's private key (actual bytes depend on real private-key):

```
signatureBytes = sign(data, <OwnPrivateKey>)
signatureBytes = "the-signature-bytes"
```

Encoded as Base64-string:

```
signature = encodeBase64(signatureBytes)
signature = "dGhlLXNpZ25hdHVyZS1ieXRlcw=="
```

Step 4

The protected endpoint is called:

```
POST http://192.168.3.17/the-api/some-protected-endpoint
```

Content sent in the request (with `data` from step 2 and `signature` from step 3):

```
{
  "data": "{\"unitId\":\"2b985ef2-e305-4fae-ad5f-20630c9109a0\",\"requestContent\":{\"some\"
:\"value\"},\"authToken\":\"d7cbb91f6367ee5a73c25999d91c1\"}",
"signature": "dGhlLXNpZ25hdHVyZS1ieXRlcw=="
}
```

Step 5

The server inspects the received content and decides if the provided authentication is valid and consistent with the provided data.

The following variables are extracted from the request content:

```
data = "{\"unitId\":\"2b985ef2…"
signature = "dGhlLXN…"
```

The `data` variable is JSON-decoded for further processing:

```
dataValue = decodeJsonString(data)
```

The following variables are extracted from the `dataValue` variable:

```
unitId = "2b985ef2-e3…"
authToken = "d7cbb91…"
requestContent = {"some":"value"}
```

The extracted `authToken` is checked for not being used yet, then it is flagged as being used.

The variable `unitId`, is extracted and the public key for that unit is looked up. Lookup that unit's public key and verify that `signature` is valid for that key and the `data` string.

The server continues with step 6, as all validation was successful. If instead any of the validations in this step would have failed, the server would respond with an error instead.

Step 6

The functionality of the protected endpoint is executed. For this the value in `requestContent` is used as the request content for further processing. Further checks and data processing on that content are performed as specified in the documentation of the protected endpoint.

```
requestContent = {"some":"value"}
```


6.4 UDP broadcasts

6.4.1 Sending UDP broadcasts

The UDP discovery packets of the different 4891-components shall at least be sent on UDP port `4891` on the applicable network interfaces.

The following grammar shall be used for building the UDP packet payload:

```
<packet>        ::=  <packet-header>
                   |  <packet-header> <spaces> <param-list>
<packet-header> ::=  "sappnet:" <packet-type>
<packet-type>   ::=  <string-without-spaces>
```

```
<param-list>     ::=  <param>
                    |  <param> <spaces> <param-list>
<param>          ::=  <string-without-spaces>
<spaces>         ::=  <space>
                    |  <space> <spaces>
<space>          ::=  " " (ASCII 0x20)
```

The `<packet-type>` depends on the concrete packed being emitted. The expected composition of the packet's `<param-list>` is coupled to the `<packet-type>` and shall always be specified alongside the concrete packet type where the sending behaviour and semantic of a packet is specified.

The whole `<packet>` sequence shall be encoded as UTF-8 string. The `<packet-type>` token shall only contain characters from these classes:

— `a` to `z` (lowercase letters);

— `A` to `Z` (uppercase letters);

— `0` to `9` (numeric digits);

— `.` (point).

6.4.2 Listening to UDP broadcasts

The 4891-components listening to UDP broadcasts shall do so on UDP port `4891`. Incoming UDP packets shall be parsed against the packet grammar as specified in 6.4.1.

The `<packet-header>` value shall be parsed as case-insensitive value, e.g. `"SappNet:SomeType"` and `"sappnet:sometype"` leading to the same result. The `<param-list>` value shall be parsed case-sensitive if not stated otherwise.

Malformed packets according to these requirements shall be ignored for the purpose of this document.

EXAMPLE 1

Well-formed packets, that lead to same result, as they are equal-content-wise:

— `"sappnet:sometype first-param second-param"`

— `"SAPPNET:SOMETYPE first-param second-param"`

— `"SAPPNET:SOMETYPE first-param second-param"`

— `"sappnet:sometype first-param second-param"`

EXAMPLE 2

Well-formed packets, that can lead to different results, as they are different content-wise:

— `"sappnet:sometype first-param second-param"`

— `"sappnet:sometype first-param SECOND-PARAM"`

— `"sappnet:sometype first-param second-param third-param"`

EXAMPLE 3

Malformed packets:

— `"xxxx:something unsupported prefix"`

— `"sappnet missing type in header"`

6.5 4891-message broker

6.5.1 General

The 4891-message broker shall implement the behaviours shown in Figure 7.

The MQTT-server shall at least provide support for using protocol version `3.1.1` (see ISO/IEC 20922).

The MQTT-server shall be configured so that other 4891-components in the same networks as the message broker can connect to it for publishing messages or subscribing to message topic filters.

If the 4891-message broker provides functionality to configure its behaviour (e.g. client authentication credentials), then such configuration functionality shall be protected from access of unauthorized users, e.g. by not making it accessible from any network shared with 4891-components or by requiring user credentials to authenticate as administrative user.

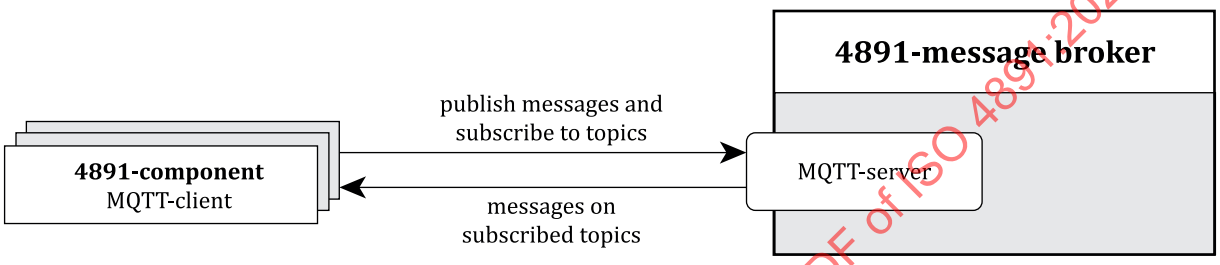


Figure 7 — Behaviour interfaces of 4891-message broker

6.5.2 Client authentication

The MQTT-server shall be configured either to allow anonymous access or with designated user credentials for all 4891-components to use when connecting to it. The manufacturer's documentation shall cover this topic and how the configuration shall be performed.

This configuration shall be reflected in the message broker connector that is configured in the 4891-service discovery (see 6.6.2.2). If user credentials are configured (and declared in the message broker connector), all 4891-components interacting with the broker shall use these credentials to authenticate against the broker.

6.5.3 Connecting to MQTT-server

To communicate with the MQTT-server of the message broker, a MQTT-client shall be used. MQTT-clients operated by 4891-components (i.e. for 4891-units) shall at least support protocol version `3.1.1` (see ISO/IEC 20922).

When connecting to the server, the clients shall use the configured authentication method, if applicable (see 6.5.2).

6.5.4 Message encoding

All 4891-messages published through the message broker on one of the defined topic patterns (see 6.5.5.2) shall be encoded in JSON format as specified in 6.2.4.

6.5.5 Publishing a 4891-message via MQTT

6.5.5.1 General

To publish 4891-messages on the message broker, a MQTT-client shall be used.

The 4891-message shall be encoded properly and published on the designated topic pattern.

6.5.5.2 Message topic

MQTT uses topic-based publication and filtering of messages on the broker.

Each 4891-message which is published on the broker shall be published on a specific topic pattern. The sending 4891-unit shall derive that topic pattern from the 4891-message's header values that is about to be published. Refer to Table 13 to build the topic string.

Table 13 — MQTT-topic string buiding patterns for 4891-messages

4891-message header condition	Delivery mode	MQTT-topic string building pattern
`DestUnitId` is not set	Broadcast	`sappnet/msg/<Type>/from/<SrcUnitId>/to-all`
`DestUnitId` is set	Single destination	`sappnet/msg/<Type>/from/<SrcUnitId>/to/<DestUnitId>`

The topic string shall always be composed from lowercase characters only. The message's type and the unit identifiers shall be converted to lowercase strings if needed.

EXAMPLE 1

A message of type `x.some.message.type` is produced by unit `87e95cbe-a5a2-452f-a4d8-5a87ceb1f6c1` with intention to broadcast it to all units interested in such a message.

The message headers are set in the following way:

— `Type` is set to `x.some.message.type`

— `SrcUnitId` is set to `87e95cbe-a5a2-452f-a4d8-5a87ceb1f6c1`

— `DestUnitId` is not set

The following topic is used to dispatch the message on the broker:

```
sappnet/msg/x.some.message.type/from/87e95cbe-a5a2-452f-a4d8-5a87ceb1f6c1/to-all
```

EXAMPLE 2

A message of type `base.signed` is produced by unit `87e95cbe-a5a2-452f-a4d8-5a87ceb1f6c1` with intention to send it to unit `32f72312-3abf-4a1f-a3c7-ebfabfd876b6`.

The message headers are set in the following way:

— `Type` is set to `base.signed`

— `SrcUnitId` is set to `87e95cbe-a5a2-452f-a4d8-5a87ceb1f6c1`

— `DestUnitId` is set to `32f72312-3abf-4a1f-a3c7-ebfabfd876b6`

The following topic is used to dispatch the message on the broker:

```
sappnet/msg/base.signed/from/87e95cbe-a5a2-452f-a4d8-5a87ceb1f6c1/to/32f72312-3abf-4a1f-a3c7-ebfabfd876b6
```

6.5.6 Subscribe to 4891-messages via MQTT

To subscribe to 4891-messages via MQTT, 4891-components shall connect their MQTT-client to the message broker's MQTT-server, if applicable.

The client shall be used to subscribe MQTT-topics of interest by using topic filter patterns. Based on the topic-building scheme in 6.5.5.2, clients can filter on different aspects of the 4891-messages when subscribing. Tables 14 and 15 show useful patterns to create topic filters based on the topic name format of 4891-messages.

To subscribe to 4891-messages targeted directly to the subscribing unit, use filters based on patterns shown in Table 14.

To subscribe to broadcasted 4891-messages, use filters based on patterns shown in Table 15.

Table 14 — MQTT-topic filter patterns for "single-destination" 4891-messages

Topic filter pattern	Selected messages
`sappnet/msg/<Type>/from/<SrcUnitId>/to/<OwnUnitId>`	Of a specific type from a specific source unit
`sappnet/msg/<Type>/from/+/to/<OwnUnitId>`	Of a specific type from any source unit
`sappnet/msg/+/from/<SrcUnitId>/to/<OwnUnitId>`	Of any type from a specific source unit
`sappnet/msg/+/from/+/to/<OwnUnitId>`	Of any type from any source unit

Table 15 — MQTT-topic filter patterns for "broadcasted" 4891-messages

Topic filter pattern	Selected messages
`sappnet/msg/<Type>/from/<SrcUnitId>/to-all`	Of a specific type from a specific source unit
`sappnet/msg/<Type>/from/+/to-all`	Of a specific type from any source unit
`sappnet/msg/+/from/<SrcUnitId>/to-all`	Of any type from a specific source unit
`sappnet/msg/+/from/+/to-all`	Of any type from any source unit

EXAMPLE 1

The topic filter to subscribe to messages of any type created by any unit with unit `619036bb-4e03-4a2f-89c4-01fb459265b8` being the single destination:

```
sappnet/msg/+/from/+/to/619036bb-4e03-4a2f-89c4-01fb459265b8
```

EXAMPLE 2

The topic filter to subscribe to messages of any type created by unit `619036bb-4e03-4a2f-89c4-01fb459265b8` with any single unit as destination:

```
sappnet/msg/+/from/619036bb-4e03-4a2f-89c4-01fb459265b8/to/+
```

EXAMPLE 3

The topic filter to subscribe to messages of any type created by any unit with destination being all units (i.e. broadcast):

```
sappnet/msg/+/from/+/to-all
```

EXAMPLE 4

The topic filter to subscribe to messages of any type created by unit `619036bb-4e03-4a2f-89c4-01fb459265b8` with destination being all units (i.e. broadcast):

```
sappnet/msg/+/from/619036bb-4e03-4a2f-89c4-01fb459265b8/to-all
```

6.6 4891-service discovery

6.6.1 General

The 4891-service discovery shall implement the behaviours shown in Figure 8.

The service discovery shall have two main functionalities:

a) a UDP-packet that is broadcast periodically containing the connectivity information to the service discovery itself;

b) a HTTP-API with a single endpoint to enable the functionality to lookup the other 4891-component's connectivity information (i.e. service connectors).

The service discovery shall provide functionality to configure service connectors for other 4891-components to lookup (see 5.2.3). That configuration functionality shall be protected from access of unauthorized users, e.g. by not making it accessible from any network shared with 4891-components or by requiring user credentials to authenticate as an administrative user.

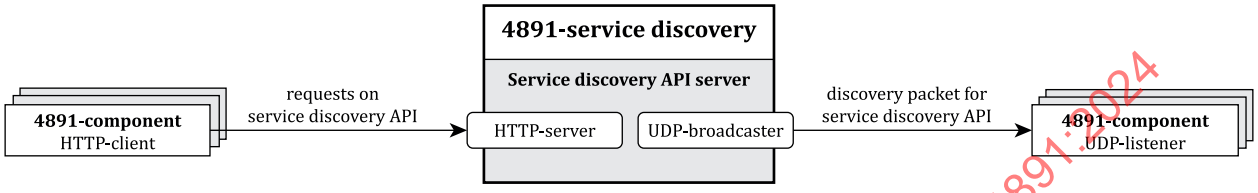


Figure 8 — Behaviour interfaces of 4891-service discovery

6.6.2 Service connectors

6.6.2.1 General

The connectivity information for the different available services (i.e. message broker, unit registry and smart gateway unit) shall be configurable (see 5.2.3).

This configuration shall only be possible by administrators.

6.6.2.2 Connector for 4891-message broker

The value type `<message-broker-connector>` shall be a dictionary with fields as specified in Table 16.

Table 16 — Connector fields for 4891-message broker

Field	Value type	Required	Description
`mqttHost`	`<string>`	Yes	Host of the MQTT-server e.g. "`192.168.10.18`"
`mqttPort`	`<integer>`	Yes	Port number of the MQTT-server e.g. `1883`
`mqttUser`	`<string>`	No	Username that units shall use when connecting to the MQTT-server. If empty or missing, then anonymous access is used instead.
`mqttPassword`	`<string>`	No	Password that units shall use when connecting to the MQTT-server. If missing, then the "empty"-password shall be used. This field is only used if non-anonymous access is used, i.e. if the field `mqttUser` is set to a non-empty value.

6.6.2.3 Connector for 4891-unit registry

The value type `<unit-registry-connector>` shall be a dictionary with fields as specified in Table 17.

Table 17 — Connector fields for 4891-unit registry

Field	Value type	Required	Description
`baseUrl`	`<base-url>`	Yes	The base URL of the unit registry. e.g. `"http://192.168.10.19:8080/unit%20registry"`

6.6.2.4 Connector for 4891-smart gateway unit

The value type `<smart-gateway-unit-connector>` shall be a dictionary with fields as specified in Table 18.

Table 18 — Connector fields for 4891-smart gateway unit

Field	Value Type	Required	Description
`unitId`	`<uuid>`	Yes	The unit identifier of the 4891-smart gateway unit in the smart application network e.g. `"7eed8707-6b3b-4c4c-afe2-321f54f4a87a"`

6.6.3 Service discovery API clients

Where applicable, 4891-units shall listen for the emitted UDP packets of the service discovery to bootstrap their service discovery connectivity.

The 4891-units shall use an HTTP-client to connect to the service discovery API. The 4891-units shall fetch the connectors from the specified endpoint to configure their local service connectivity for reaching the other central 4891-components.

The 4891-units shall refresh this connectivity information periodically. Manufacturers are advised to implement caching and refreshing strategies. The following points shall be taken into consideration when designing such strategies:

— on a new network setup, it is possble that not all connectors are available from the beginning (a few having `null` values), so units shall expect connectors to become available over time;

— units shall expect that connectors can change over time;

— connectors shall be refreshed periodically and when joining new networks.

6.6.4 Service discovery API server

6.6.4.1 General

The 4891-service discovery shall run an HTTP-server implementing the API endpoints specified in 6.6.4.2. These endpoints shall be reachable by URLs built by concatenating `ServiceDiscoveryApiBaseUrl` and the endpoint specific path-suffix.

6.6.4.2 Endpoint GetServiceConnectors

The single endpoint of the service discovery responds with connectivity information for the available services in the smart application network (see 6.6.2 and 5.2.3).

A `null` value for a connector can mean the following:

— the component is not available on the used network;

— the component has not yet been configured in the service discovery.

Table 19 specifies the signature of this endpoint.

Table 19 — HTTP-signature of endpoint GetServiceConnectors

Method	GET	
Path	<ServiceDiscoveryApiBaseUrl>/services	

Response	**Content**	{
		"messageBroker": <null> | <message-broker-connector>
		"unitRegistry": <null> | <unit-registry-connector>
		"smartGatewayUnit": <null> | <smart-gateway-unit-connector>
		}

6.6.5 Service discovery API discovery packet

The service discovery shall emit UDP packets on the networks shared with the other 4891-components.

The packet shall be sent periodically once every 5 s to 10 s.

The packet shall be sent on broadcast addresses of the applicable network interfaces.

The packet shall contain the URL to the discovery API that is reachable on the network on which the packet has been sent.

The packet shall use the basic specification described in 6.4 with the following additions:

```
<packet-type> ::=  "sd"
<param-1>     ::=  <ServiceDiscoveryApiBaseUrl>

<ServiceDiscoveryApiBaseUrl> ::=  <base-url>
```

6.6.6 Service discovery API examples

In the following Examples 1 and 2, a 4891-service discovery broadcasts UDP packets in the network to disclose its API connectivity information. With that connectivity information, a 4891-unit is invoking HTTP-endpoints on the service discovery API. The following service discovery API base URL is used for all examples:

```
http://192.168.2.123:1234/base-path/to/discovery-api
```

EXAMPLE 1

The following UDP packet is broadcast every 5 seconds on some network interface by the service discovery:

```
sappnet:sd http://192.168.2.123:1234/base-path/to/discovery-api
```

A recipient of these packets parses it and identifies the following information from it:

— The packet header `sappnet:sd` indicates that this is a smart application network discovery packet for service discovery API (see 6.6.5).

— The first parameter provides the base URL for the service discovery API server:

```
http://192.168.2.123:1234/base-path/to/discovery-api
```

The recipient can now perform requests against the service discovery API by using the received base URL for building the endpoint URLs.

EXAMPLE 2

A unit has previously picked up the service discovery API base URL (e.g. by listening for UDP packets as in Example 1):

```
http://192.168.2.123:1234/base-path/to/discovery-api
```

The unit opens a TCP socket to the service discovery API host (192.168.2.123 on port 1234) and sends a HTTP-request to lookup the available services:

```
GET /base-path/to/discovery-api/services HTTP/1.1
Host: 192.168.2.123
Accept: application/json
```

The service discovery responds on that socket with an HTTP-response containing the available services encoded as JSON:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "messageBroker": {
    "mqttHost": "192.168.42.7",
    "mqttPort": 1883,
    "mqttUser": "user",
    "mqttPassword": "password"
  },
  "unitRegistry": {
    "baseUrl": "http://192.168.41.8/some-prefix"
  },
  "smartGatewayUnit": {
    "unitId": "b3792662-03fe-4466-a4e0-8f9d567a5e56"
  }
}
```

Now the requesting unit knows how to connect to the other central 4891-components that are available in the same network.

6.7 4891-unit registry

6.7.1 General

Units that want to participate in the smart application network shall register themselves at the 4891-unit registry.

The unit registry shall provide the following interface functionality for communication with the units (see Figure 9). The implementation shall operate a HTTP-server, whose connectivity information shall be made available via the service discovery API (see 6.6.2.3).

The unit registry shall provide functionality to configure root certificates as described in 6.10. That configuration functionality shall be protected from access of unauthorized users, e.g. by not making it accessible from any network shared with 4891-components or by requiring user credentials to authenticate as an administrative user.

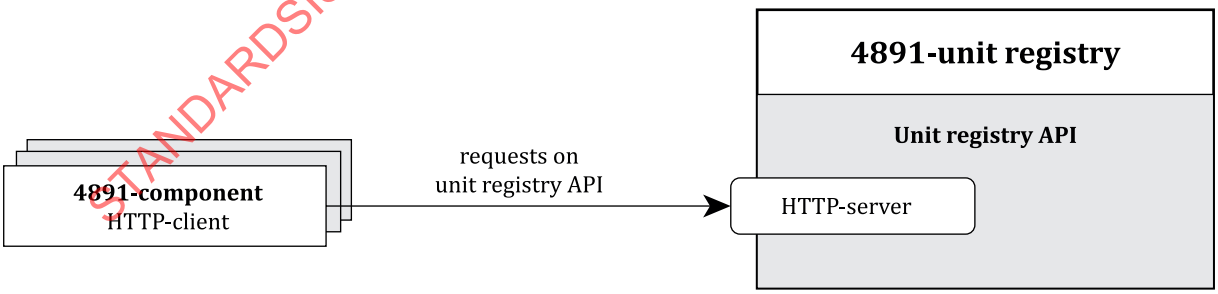


Figure 9 — Behaviour interfaces of 4891-unit registry

6.7.2 Tracking of unit information

6.7.2.1 General

The unit registry shall track the following information about 4891-units registered in the smart application network:

— the unit's name (received from unit while registering);

— the unit's public key (received from unit while registering);

— the unit's identifier (assigned by unit registry);

— the unit's unit certificate (issued by unit registry);

— the unit's timestamp of registration (“registered at”);

— the unit's timestamp of last known activity (“last activity at”).

The “last activity at” timestamp information shall be gathered by the 4891-unit registry. At least the following methods shall be used by the unit registry to gather this information.

6.7.2.2 Direct communication with the unit

Whenever there is direct communication between a unit and the unit registry, i.e. by successfully using a protected endpoint (see 6.3.6), the “last activity at” information for that unit shall be updated to the current time.

6.7.2.3 Observed messages on message broker

The unit registry shall listen to messages published on the message broker and inspect the `SrcUnitId` and `CreatedAt` headers of those. For messages of type `base.signed` (see 6.10.5), the contained messages shall also be inspected. With this gathered information, the registry shall update the “last activity at” information of affected units.

6.7.3 Unit registry API clients

The unit registry API shall be used by all 4891-units to at least lookup root certificates and to register themselves as a proper unit in the smart application network.

For interacting with the API, the unit shall use an HTTP-client.

6.7.4 Unit registry API server

6.7.4.1 General

The 4891-unit registry shall run an HTTP-server implementing the API endpoints specified in 6.7.4. These endpoints shall be reachable by URLs built by concatenating `UnitRegistryApiBaseUrl` and the endpoint specific path-suffix.

6.7.4.2 Endpoint “GetActiveRootCertificates”

This endpoint provides access to the root certificates that shall be considered active when used in the context of unit certificate validation (see 5.7 and 6.10).

Table 20 specifies the signature of this endpoint.

Table 20 — HTTP-signature of endpoint “GetActiveRootCertificates”

Method	GET	
Path	<UnitRegistryApiBaseUrl>/root-certificates	

Request	**Content**	{
		“certificates”: <list of certificate-pem>
		}

6.7.4.3 Endpoint “RegisterUnit”

This endpoint shall perform the unit registration and unit certificate issuing logic described in 6.10.4. If the registration is successful, the provided unit name shall be stored as meta information for that unit.

This endpoint shall respond with the identifier and unit certificate associated to the provided public key.

Granted a 4891-unit keeps its private key which is related to the public key from a valid unit certificate, it shall be considered to be a registered unit of the network. It does not matter if that unit loses physical connectivity from the network or if it disconnects and reconnects again later.

Table 21 specifies the signature of this endpoint.

Table 21 — HTTP-signature of endpoint “RegisterUnit”

Method	POST	
Path	<UnitRegistryApiBaseUrl>/units	

Request	**Content**	{
		“name”: <string>
		“publicKey”: <publickey-pem>
		}

Response	**Content**	{
		“id”: <uuid>
		“certificate”: <certificate-pem>
		}

6.7.4.4 Endpoint “ModifyUnitInformation”

This endpoint shall be protected by unit authentication as described in 6.3.6.

This endpoint shall be used by 4891-units for modifying information associated to them in the unit registry.

The unit registry shall validate that the unit identifier provided in the path matches the unit identifier that has been specified for authenticating on this protected endpoint (see 6.3.6.4). If that validation fails, the request shall fail with error code FORBIDDEN.

The unit registry shall update the information associated to the unit by the provided values in this request.

Table 22 specifies the signature of this endpoint.

Table 22 — HTTP-signature of endpoint “ModifyUnitInformation”

Method	PATCH
Path	<UnitRegistryApiBaseUrl>/units/<UnitId>

Protected request content	{
	“name”: <string>
	}

6.7.4.5 Endpoint “GetUnitInformation”

This endpoint allows information about a specific unit to be looked up by providing its unit identifier in the endpoint's path.

The unit registry shall lookup the information from its database for returning it as response.

If no unit can be found by the provided identifier, a HTTP-response with status `404` shall be returned.

Table 23 specifies the signature of this endpoint.

Table 23 — HTTP-signature of endpoint “GetUnitInformation”

Method	`GET`	
Path	`<UnitRegistryApiBaseUrl>/units/<UnitId>`	
Response	**Content**	`<unit-info>`

The value type `<unit-info>` shall be a dictionary with fields as specified in Table 24.

Table 24 — Fields of dictionary type “<unit-info>”

Field key	Value type	Required	Description
`id`	`<uuid>`	yes	The unit's unit identifier.
`name`	`<string>`	yes	The name that the unit has given itself.
`certificate`	`<certificate-pem>`	yes	The latest unit certificate.
`registeredAt`	`<timestamp>`	yes	The timestamp of unit registration.
`lastActivityAt`	`<timestamp>`	yes	The timestamp of latest known activity of that unit.

6.7.4.6 Endpoint “GetListOfUnitInformation”

This endpoint returns a page from the unit list. The full list shall contain all units registered at the unit registry. The returned page of the list shall be built by following these steps in this order:

a) ensure the initial result set contains all units registered at the unit registry (“total set”);

b) apply optional filtering (`registeredAfter` and `activityAfter`) to that set (“filtered set”);

c) apply sorting (`sort` and `sortDir`), resulting in an ordered list (“filtered list”);

d) extract a sub-list by starting at index `offset` (0-based) taking up to `limit` units (“result page”).

The requesting component shall use the limit parameter only as a hint to indicate a desired quantity of units from the API. The unit registry may apply an upper bound to the `limit` parameter, overriding it with a smaller value than provided by the component. The unit registry may return a lower number of units than has been requested. The unit registry shall not return more units than has been requested.

Table 25 specifies the signature of this endpoint.

Table 25 — HTTP-signature of endpoint “GetListOfUnitInformation”

Method	`GET`			
Path	`<UnitRegistryApiBaseUrl>/units`			
Query Params	**Param name**	**Value type**	**Description**	
	`registeredAfter`	`<timestamp>`	Filters the result set by only including units that had been registered after provided timestamp.	
	`activityAfter`	`<timestamp>`	Filters the result set by only including units that have a `lastActivityAt` timestamp of at least provided value.	
	`offset`	`<integer>`	the 0-based index after sorting for the first result element (default: `0`)	
	`limit`	`<integer>`	maximum number of units to be returned (default: `100`)	
	`sort`	`<enum>`	Defines the field to use for sorting the unit list before pagination is applied. Allowed values are: — `registeredAt` (default) — `lastActivityAt` — `name`	
	`sortDir`	`<enum>`	Defines the direction of sorting of the unit list. Allowed values are: — `asc` (ascending, default) — `desc` (descending)	
Response	**Content**	**Param name**	**Value type**	**Description**
		`total`	`<integer>`	Number of units registered in the registry ignoring the filter parameters (length of “total set”)
		`filtered`	`<integer>`	Number of units registered in the registry matching the filter parameters (length of “filtered set”)
		`units`	`<list of unit-info>` (see Table 24)	List of units from the “filtered list” starting at requested `offset` index.

6.7.5 Unit registry API examples

In the examples 1 to 6 below, a unit has already received the connector information for the unit registry API from the service discovery. With the base URL of the unit registry API, the unit can now invoke the available HTTP-endpoints. The following unit registry API base URL is used for all examples:

```
http://192.168.41.8/some-prefix
```

EXAMPLE 1

The unit sends a HTTP-request to the “get root certificates”-endpoint on the unit registry API to get the valid root certificates:

```
GET http://192.168.41.8/some-prefix/root-certificates
```

The unit registry responds with HTTP-status 200 and the available root certificates (two in this example) as content:

```
{
  "certificates": [
    "---BEGIN CERT…",
    "---BEGIN CERT…"
  ]
}
```

The unit updates its local cache with this received certificates, by storing them in a local database, replacing previously fetched root certificates. With those root certificates the unit is now able to validate unit certificates.

EXAMPLE 2

The unit generates a cryptographic key-pair to use for encryption and digital signatures (see 6.10) and sends the public key and a name for the unit to the registration endpoint:

```
POST http://192.168.41.8/some-prefix/units
{
  "name": "Some Unit Name",
  "publicKey": "---BEGIN PUBLIC KEY…"
}
```

The unit registry responds with HTTP-status 200, an identifier for the registering unit (“unit id”) and the unit's certificate (signed by one of the root certificates of the unit registry):

```
{
  “id”: “fd8c64fd-8bc9-47ae-abdc-cbc88d8feea7”,
  “certificate”: “---BEGIN CERT…”
}
```

With this, the unit is now known to the registry and other units can lookup the unit's certificate, too.

EXAMPLE 3

The unit wants to modify its own information via endpoint `ModifyUnitInformation`. Because that is a protected endpoint, it is used in the way described in 6.3.6.4.

The unit calls the endpoint RequestAuthToken on the API first, to receive a one-time usable token:

```
POST http://192.168.41.8/some-prefix/auth-token
```

The unit bundles the received token, its unit id and the protected request content of the endpoint into a JSON object. Then it encodes that object into a string, signs it with its own private key and adds both the string and the signature to the request as content:

```
PATCH http://192.168.41.8/some-prefix/units
{
  "data": "{\"unitId\":\"fd8c64fd-\",\"authToken\":\"…\",\"requestContent\":{\"name\":\"My
new Unit Name\"}}",
  "signature": "YmluYXJ5IHNpZ25hdHVyZSBkYXRhIC4uLg…"
}
```

The endpoint will validate the signature (with `data` string and `unitId`) and the `authToken` (one-time use) provided before performing the logic.

With this, the unit has changed its meta-information on the unit registry.

EXAMPLE 4

Requesting first page of size 10 of registered units via unit registry API:

```
GET http://192.168.41.8/some-prefix/units?offset=0&limit=10
```

Requesting third page of size 10 of registered units via unit registry API:

```
GET http://192.168.41.8/some-prefix/units?offset=20&limit=10
```

Requesting first page of size 25 of registered units that were active after 2021-05-11 11:35:00 UTC via unit registry API:

```
GET http://192.168.41.8/some-prefix/units?limit=25&activityAfter=2021-05-11T11:35:00Z
```

EXAMPLE 5

Example of a unit registry API response on endpoint `GetListOfUnitInformation` for request with page size 3 and without filters:

```
{
      “total”: 175,
      “filtered”: 175,
      “units”: [
      {
                “id”: “89562e…”,
                “name”: “Old Legacy Device”,
                “certificate”: “---BEGIN CERT…”,
                “registeredAt”: “2020-03-03T12:00:01Z”,
                “lastActivityAt”: “2020-09-05T13:14:16Z”
      },
      {
                “id”: “6ba2a6cb…”,
                “name”: “Some Unit”,
                “certificate”: “---BEGIN CERT…”,
                “registeredAt”: “2020-03-23T12:56:13Z”,
                “lastActivityAt”: “2021-05-11T17:35:00Z”
      },
      {
                “id”: “aa6a…”,
                “name”: “Sensor X”,
                “certificate”: “---BEGIN CERT…”,
                “registeredAt”: “2020-03-24T09:12:07Z”,
                “lastActivityAt”: “2021-05-12T18:24:13Z”
      }
    ]
 }
```

EXAMPLE 6

Example of a unit registry API response on endpoint `GetListOfUnitInformation` for request with page size 3 and with recent-activity filter:

```
{
  "total": 175,
  "filtered": 23,
  "units": [
    {
      "id": “6ba2a6cb…",
      "name": “Some Unit",
      "certificate": "---BEGIN CERT…",
      "registeredAt": "2020-03-23T12:56:13Z",
      "lastActivityAt": "2021-05-11T17:35:00Z"
    },
    {
      "id": "aa6a…",
      "name": "Sensor X",
      "certificate": "---BEGIN CERT…",
      "registeredAt": "2020-03-24T09:12:07Z",
      "lastActivityAt": "2021-05-12T18:24:13Z"
    },
    {
      "id": "4a43…",
      "name": "Human Interface Device",
      "certificate": "---BEGIN CERT…",
      "registeredAt": "2020-04-04T16:37:13Z",
      "lastActivityAt": "2021-05-12T09:35:07Z"
    }
  ]
}
```

6.8 4891-unit

All 4891-units shall implement the behaviours shown in Figure 10. Those behaviours are described in 6.5, 6.6, 6.7 and 6.9.

The behaviour “direct messaging with units” (see 6.9) is optional. Manufactures may skip that behaviour from their implementation.

Additional behaviours for 4891-units are specified in Annexes A and B.

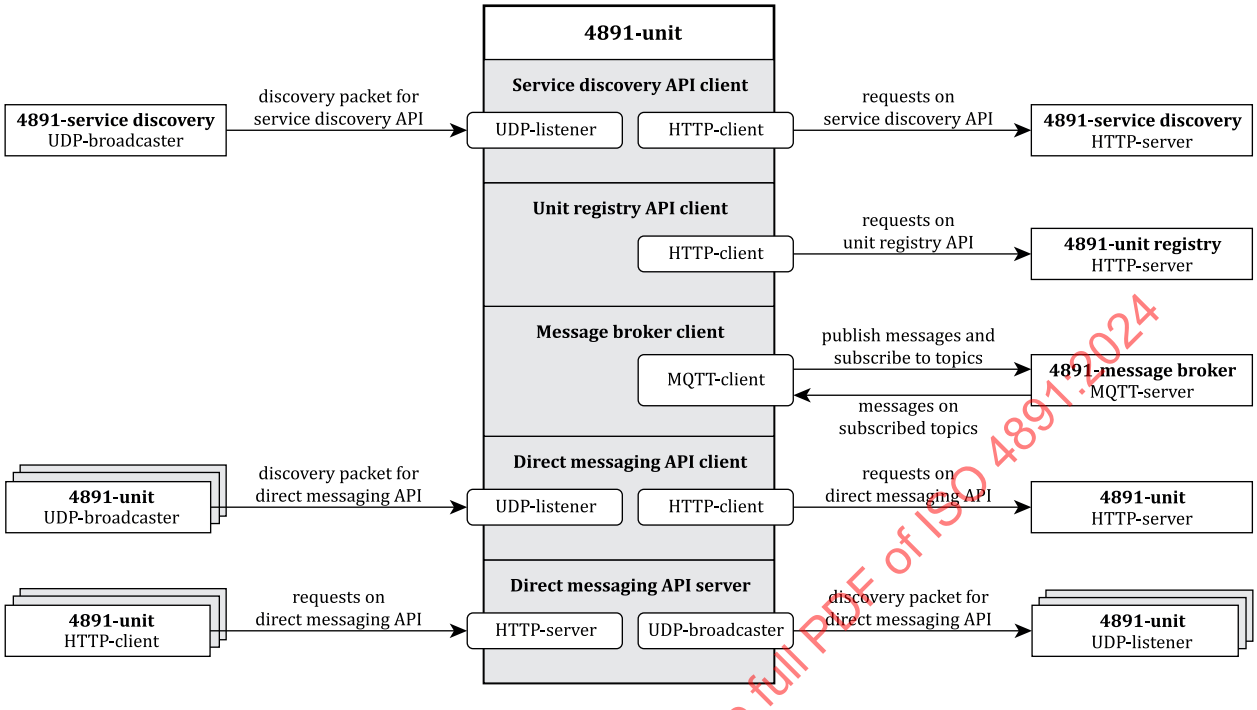


Figure 10 — Behaviour interfaces of 4891-unit

6.9 Direct messaging API

6.9.1 General

4891-units that support the receiving side of the direct messaging functionality shall provide the following interface:

— an HTTP-API to accept incoming 4891-messages, i.e. the direct messaging API;

— broadcasting of an UDP discovery packet for the direct messaging API.

With this interface, those units will be able to accept incoming messages for further processing, i.e. by consuming them for itself (receiving unit is the message's destination) or by relaying it to the destination unit. If the unit accepts a message to be relayed to another unit, it shall do one of the following:

— send that message to the message broker;

— send that message directly to the designated destination unit if that unit supports direct messaging functionality and is reachable by the unit;

— send it to another unit which is not the designated recipient unit for further relaying.

Implementations shall prevent endlessly relaying of messages in a circular way, i.e. they shall have a concept of computing a route to the designated recipient unit in some way if this option is used. Adding an expiry timestamp to the relayed message (`ExpiresAt` message header) may also be a helpful mitigation.

6.9.2 Direct messaging API clients

The clients of the direct messaging API of a 4891-unit are other 4891-units that support the sending side of the direct messaging functionality.

Where applicable, those units shall listen for the emitted UDP packets of the direct messaging API of other units, to bootstrap their potential direct messaging connectivity to other units. Alternatively, it is also allowed to configure the direct messaging API base URL to another unit in other ways.

For using the API, the sending 4891-units shall use an HTTP-client to connect to the receiving unit's direct messaging API, by utilizing its API base URL.

Sending a 4891-message directly to another unit can be desirable in different situations, i.e.:

— The receiving unit is the designated destination of a message, so the message can be sent to it directly rather than indirectly via a 4891-message broker.

— The sending unit is not connected to the 4891-message broker, so the receiving unit can be used for message relaying to deliver that message at a later stage.

Units may use the direct messaging API for automatically sending messages to another unit or by explicitly triggering that functionality on user interaction. When using fully automated sending logic, manufacturers shall implement logic to prevent or mitigate circular relaying of messages.

6.9.3 Direct messaging API server

6.9.3.1 General

4891-units supporting the receiving-side of the direct messaging functionality shall run an HTTP-server implementing the API endpoints specified in 6.9.3. These endpoints shall be reachable by URLs built by concatenating `DirectMessagingApiBaseUrl` and the endpoint specific path-suffix.

6.9.3.2 Endpoint “GetUnitInformation”

This endpoint allows looking up information about the 4891-unit that is operating the direct messaging API server. This endpoint can be used as an alternative for looking up that unit's information instead of using endpoints on the unit registry API.

The server shall respond with information about the unit operating the server, including:

— the unit's identifier;

— the unit's name;

— the unit's certificate that it received from the unit registry.

If the unit had not yet registered itself (i.e. identifier and certificate not available), then the server shall respond with error code `NOT_FOUND`.

Table 26 specifies the signature of this endpoint.

Table 26 — HTTP-signature of endpoint “GetUnitInformation”

Method	`GET`	
Path	`<DirectMessagingBaseUrl>/unit`	

Response	**Content**	`{`
		`  “id”: <uuid>`
		`  “name: <string>`
		`  “certificate”: <certificate-pem>`
		`}`

6.9.3.3 Endpoint “SendMessage”

This endpoint allows sending a single message to the 4891-unit operating the direct messaging API server.

If the receiving unit has already registered itself (i.e. identifier and certificate are available), then the received message shall be forwarded into the processing logic specified in 6.11. If the unit has not yet registered itself, then the received message shall either be ignored, or stored until the unit has registered itself.

The server shall respond with HTTP-status `204` (no content) independent of the specific message received.

Table 27 specifies the signature of this endpoint.

Table 27 — HTTP-signature of endpoint "SendMessage"

Method	`POST`	
Path	`<DirectMessagingBaseUrl>/messages`	

Request	**Content**	`{`
		`  "message": <message>`
		`}`

Response	**Status**	`204` (no content)

6.9.4 Direct messaging API discovery packet

4891-units that run a direct messaging API server shall emit UDP packets on their network interfaces that are intended to interact with the smart application network, e.g. where the unit wants to be contactable by other units.

The packet shall be emitted periodically once every 5 s to 60 s. If the unit's direct messaging API server is only running for short amounts of time (i.e. by explicitly switching it on and off again on demand) it is allowed to emit the packets more frequently than every 5 s in that period.

The packet shall not be sent in any of these situations:

— The unit has not yet registered at the unit registry, i.e. does not have a unit identifier.

— The direct messaging server is not running.

The packet shall be sent on broadcast addresses of the applicable network interfaces. Additional ways to distribute the discovery packet are allowed.

The packet shall contain the sending unit's identifier and the base URL for reaching the unit's direct messaging API on the network where the packet has been sent.

The packet shall use the basic specification described in 6.4 with the following additions:

```
<packet-type> ::=  "dm"
<param-1>     ::=  <UnitId>
<param-2>     ::=  <DirectMessagingApiBaseUrl>

<UnitId>                    ::=  <uuid>
<DirectMessagingApiBaseUrl> ::=  <base-url>
```

6.9.5 Direct messaging API examples

In the Examples 1 and 2, a 4891-unit broadcasts UDP packets in the network to disclose its direct messaging API connectivity information. With that connectivity information, another 4891-unit is invoking HTTP-endpoints on that API. The following direct messaging API base URL is used for all examples:

```
http://192.168.2.40:8080/my-dm-api
```

EXAMPLE 1

The following UDP packet is broadcast every 10 s on some network interface by a unit which operates a direct messaging API server:

```
sappnet:dm 1b570fa5-e666-43fb-a8af-1f29f9a98bb1 http://192.168.2.40:8080/my-dm-api
```

A recipient of these packets parses it and identifies the following information from it:

— The packet header `sappnet:dm` indicates that this is a smart application network discovery packet for direct messaging API (see 6.9.4).

— The first parameter provides the identifier of the unit that operates the direct messaging API server, i.e. the sender of the packet: `1b570fa5-e666-43fb-a8af-1f29f9a98bb1`

— The second parameter provides the base URL for the direct messaging API server: `http://192.168.2.40:8080/my-dm-api`

The recipient can now perform requests against the direct messaging API server of the sending unit by using the received base URL for building the endpoint URLs.

EXAMPLE 2

A unit has previously picked up the base URL to a direct messaging API server from a discovery-packet of another unit as described in Example 1.

The unit that picked up the packet can now lookup the broadcasting unit's information and certificate, e.g. to update its locally cached information. This can be done either via the central unit registry API (see 6.7.4.5) or like in this example via the direct messaging API server operated by the broadcasting unit (see 6.9.3.2):

```
GET http://192.168.2.40:8080/my-dm-api/unit
```

With a valid unit certificate for that unit available, a message can be created and encrypted for that unit. Sending the message to the unit can then be done via message broker or like in this example via the direct messaging API server operated by the broadcasting unit (see 6.9.3.3):

```
POST http://192.168.2.40:8080/my-dm-api/messages
```

6.10 Trusted communication

6.10.1 General

Trusted communication between 4891-units shall be enabled by using a PKI, with trusted certificates and asymmetric cryptography to encrypt and digitally sign data.

6.10.2 Public key infrastructure

The 4891-unit registry shall act as CA, representing the root of trust in the smart application network.

Two types of certificates are of interest in this PKI:

— Root certificates: these are used by the CA to sign unit certificates issued by the CA. The other 4891-components can look-up these root certificates for validating the integrity of any CA-issued certificates.

— Unit certificates: these are issued by the CA for 4891-units registering themselves. They prove the association of a unit identifier to a key-pair owned by the unit. If a matching private key for the public key is held by the unit, that unit is considered to be registered with the assigned unit id.

The 4891-unit registry shall be configured with at least one root certificate (see 6.10.3). Those root certificates shall be used for signing unit certificates for all 4891-units that register via its unit registry API.

4891-units that support trusted communication shall use their own private key and other unit's unit certificates for encrypting/decrypting and signing/verification. 6.10 describes the required functionality of 4891-units and the 4891-unit registry with regards to performing these operations and handling of the certificates.

All certificates shall use X.509 as the data format.

For exchanging certificate and key information, PEM encoding shall be used.

Table 28 lists algorithms that shall be utilized for cryptographic operations related to 4891-messages and certificates.

Table 28 — Algorithms for use in cryptographic operations

Cipher algorithms	Hashing algorithms
— RSA	— SHA-256
— DSA	— SHA-384
— ECDSA	

4891-components that perform one of the following operations, shall support all algorithms listed in Table 28:

— verifying root certificates;

— verifying user certificates;

— verifying signatures of 4891-messages;

— decrypting 4891-messages.

4891-components that perform one of the following operations, shall support at least one cipher algorithm and at least one hashing algorithm listed in Table 28:

— generating a key-pair (e.g. unit registration);

— issuing root certificates;

— issuing user certificates;

— signing 4891-messages;

— encrypting 4891-messages.

6.10.3 Root certificates

6.10.3.1 General

The root certificates shall be considered as trustworthy, making them the “root of trust” in the PKI.

Root certificates shall have these properties:

— the subject's common name (`CN`) shall be set to value `sappnet:ureg`

— validity duration shall be picked with a pragmatic approach, trading-off these factors:

  — shorter validity duration for higher security,

  — longer validity duration for more convenient use (less user interaction required).

6.10.3.2 Configuring root certificates

The unit registry shall be configurable with root certificates in one of the following ways:

— by manually importing valid X.509 certificates and matching private keys for subject public keys of those certificates;

— by implementing a functionality to create key-pairs and create self-signed X.509 certificates for them.

In case a certificate is required to be deleted and replaced with a different root certificate, it is recommended to perform these steps:

a) add/create an additional root certificate that is valid immediately;

b) select the new certificate as the main certificate after 8 days, either manually or automatically (see 6.10.3.3);

c) remove the old certificate after waiting for another 8 days, until then the new certificate is likely to be picked up and fully understood by all 4891-components.

If these steps are not followed, some 4891-components can run into communication problems based on invalid certificates in case they cache root certificates.

6.10.3.3 Selecting main root certificate

If the 4891-unit registry has been configured with multiple root certificates, one of those shall be flagged as being the main certificate.

The selection may be done by manual configuration as described in the manufacturer's documentation, or automatically. Manufacturers shall consider the following points when implementing a selection algorithm:

a) never select certificates that are not valid at the time of selection (i.e. not valid before/after constraints);

b) filter out certificates that have a “not valid before” value that is more recent than 8 days ago;

c) take the certificate with the most recent value in “not valid before”;

d) if that algorithm does not find any results, skip step b) and retry.

6.10.3.4 Fetching root certificates

4891-components that support trusted communication shall fetch the root certificates directly from the 4891-unit registry via the unit registry API (see 6.7.4.2).

6.10.3.5 Caching root certificates

Root certificates may be persisted locally (cached) after being.

Such cached root certificates shall be flagged as “potentially outdated” after being cached for longer than 7 days without being refreshed.

Potentially outdated certificates can still be used for validation of trust-chains, but it is recommended to refresh the list of cached certificates before they become potentially outdated and also to pick up any new root certificates that have been configured.

6.10.3.6 Trusting root certificates

4891-components shall not trust root certificates which have been received automatically.

4891-components that support trusted communication shall provide functionality to review received root certificates to explicitly flag them as either trusted or untrusted.

It is advised for installations of the smart application network to have an onboard documentation prominently on display to list the official root certificates in use in the network. That documentation can then be reviewed by users operating 4891-units or administrating the 4891-components to decide if a given root certificate is trustworthy.

The 4891-components shall store the decision of the user on a per-root-certificate base and respect it for further processing (i.e. trusting the certificate or not).

6.10.3.7 Validating root certificates

All root certificates fetched from the unit registry shall be considered as being active, while their timestamp constraints (i.e. “not valid before”, “not valid after”) are not broken.

All root certificates that had previously been considered active, but that are not listed in the response of the unit registry anymore, shall be considered being revoked.

Only root certificates that are active and that have been flagged as trusted (see 6.10.3.6) shall be considered valid.

6.10.4 Unit certificates

6.10.4.1 General

Unit certificates bind a unit identifier to a key-pair whose private key is only known by the 4891-unit represented by that identifier. By using that key-pair for asymmetric cryptography (see 5.7, 6.10.5 and 6.10.6), trusted communication and authentication can be achieved between 4891-units and other 4891-components.

Unit certificates shall have the following properties:

— the subject's common name (`CN`) shall be set to a value of format `sappnet:unit:<UnitId>`;

— the subject's public key shall be set to the unit's public key;

— the issuer's certificate shall be the main root certificate

— signed by the private key matching the used root certificate;

— validity duration shall be picked with a pragmatic approach, trading-off these factors:

  — shorter validity duration for higher security,

  — longer validity duration for better stability (less often renewals required).

EXAMPLE

A unit’s identifier and the related common name field in the subject of its unit certificate:

```
identifier:   d7d667df-e0ba-4da8-a386-926ec2d2c338
subject:      CN=sappnet:unit:d7d667df-e0ba-4da8-a386-926ec2d2c338
```

6.10.4.2 Issuing unit certificate (unit registration)

Unit certificates shall be issued by the 4891-unit registry as part of the unit registration process. The main root certificate at the time of issuing shall be used for signing the new unit certificate.

The 4891-unit shall generate a new key-pair and send the public key to the 4891-unit registry for registration. The unit registry shall generate a new unique unit identifier and bind it to the received public key by generating and signing a new unit certificate containing the unit identifier and received public key. That certificate shall be signed by the main root certificate.

If the received public key is already known by the unit registry, then no new unit identifier and certificate shall be issued.

If no main root certificate can be selected, then the process shall fail with an error.

4891-units shall implement logic based on Figure 11.

4891-unit registry shall implement logic based on Figure 12.

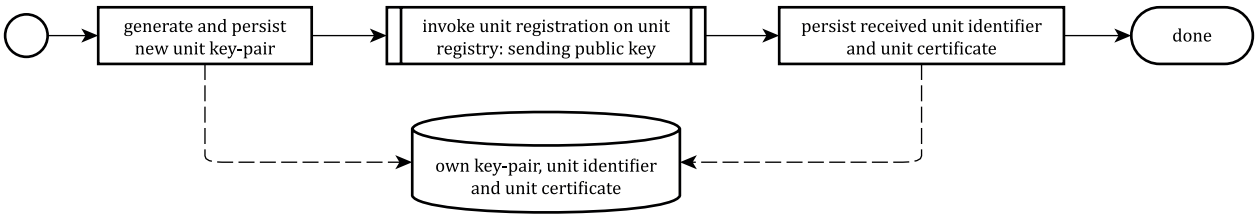


Figure 11 — Unit registration logic for 4891-unit

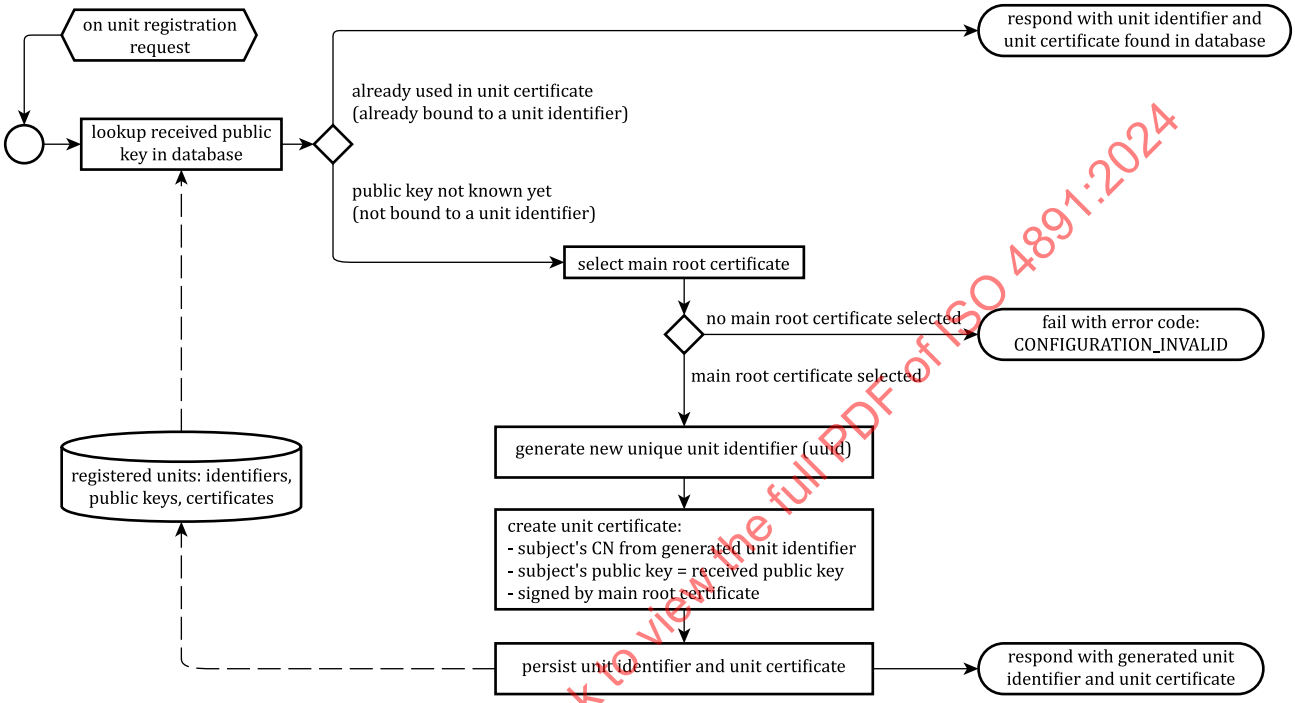


Figure 12 — Unit registration logic for 4891-unit registry

6.10.4.3 Re-issuing unit certificates

The 4891-unit registry shall re-issue previously issued unit certificates whenever the main root certificate changes. Manufacturers may implement this as an automatic process happening as a side-effect of the certificate change, or by re-issuing whenever a unit certificate is looked up.

6.10.4.4 Fetching unit certificates

4891-components that support trusted communication may gather unit certificates in any of the following ways:

— lookup unit certificates via unit registry API;

— lookup unit certificates via direct messaging API of units;

— collect unit certificates contained in received 4891-messages (e.g. `base.signed`).

6.10.4.5 Caching unit certificates

Unit certificates may be persisted locally (cached) after being fetched from the unit registry or any other source.

It is possible for a unit certificate to become invalid, i.e. if the root certificate becomes invalid (expired or revoked). In such cases, the unit certificate should be refreshed by fetching a new version of it (see 6.10.4.4 and 6.7.4.5).

When receiving different unit certificates for the same unit identifier, the recipient shall favour the certificate that has been issued more recently, if the used root signing key is from a trusted root certificate.

6.10.4.6 Validating unit certificates

A unit certificate shall be considered valid if the following criteria are fulfilled:

a) the unit certificate is signed by one of the root certificates;

b) the root certificate was not revoked;

c) the root certificate was valid at the unit certificate's issuing timestamp.

A unit certificate shall be considered to match a given unit identifier, if that unit identifier is part of the certificate's subject common name (see 6.10.4.1).

6.10.5 Signing data (digital signatures)

6.10.5.1 General

Digital signatures shall be used whenever the integrity of data is important for an implemented use case. That allows the recipient to check if the received data has been modified on transport. The following steps describe the general process to digitally sign data and check its integrity upon receipt:

a) the data to secure is digitally signed with the sender's private key, producing a digital signature;

b) the data and the digital signature are sent to the recipient;

c) the recipient looks up the public key of the sender (e.g. from a trusted certificate);

d) the recipient uses that public key to validate that the data and digital signature are consistent.

6.10.5.2 Message type “base.signed”

To digitally sign one or multiple 4891-messages, they shall be wrapped into a message of type `base.signed`. This shall be done by following these steps:

a) build a list of the 4891-messages to be signed;

b) encode that list of messages into a JSON-encoded string;

c) compute a digital signature of that JSON-encoded string with the unit's own private key;

d) create a new message of type `base.signed` containing:

   — the JSON-encoded string containing the original messages;

   — the computed signature;

   — optionally, add the unit certificate of the unit that signed the messages.

Table 29 specifies the composition for this message type.

Table 29 — Composition of 4891-message type "base.signed"

Message type	`base.signed`	
Data part 1	**Encoding**	`TEXT`
	Data	An array of 4891-messages serialized as a JSON-encoded string. This string value was used as payload that has been signed, producing the signature in data part 2.
Data part 2	**Encoding**	`BINARY`
	Data	The signature data.
Data part 3 (optional)	**Encoding**	`TEXT`
	Data	The unit certificate of the unit that has signed the message, encoded as PEM-string.

Unpacking the contained messages shall be done by decoding the JSON-encoded data in data part 1 in Table 29 to receive the list of original 4891-messages.

Any 4891-component receiving such a message can validate that the message has not been modified once it is signed by the unit creating the message. The validation shall be performed by the steps shown in Figure 13, consisting of:

e) looking up a valid the unit certificate for the message's `SrcUnitId` header (from data part 3 Table 29, cache or 4891-unit registry);

g) validating that the signature in data part 2 Table 29 is consistent for the unit certificate's public key on the JSON-string in data part 1 Table 29.

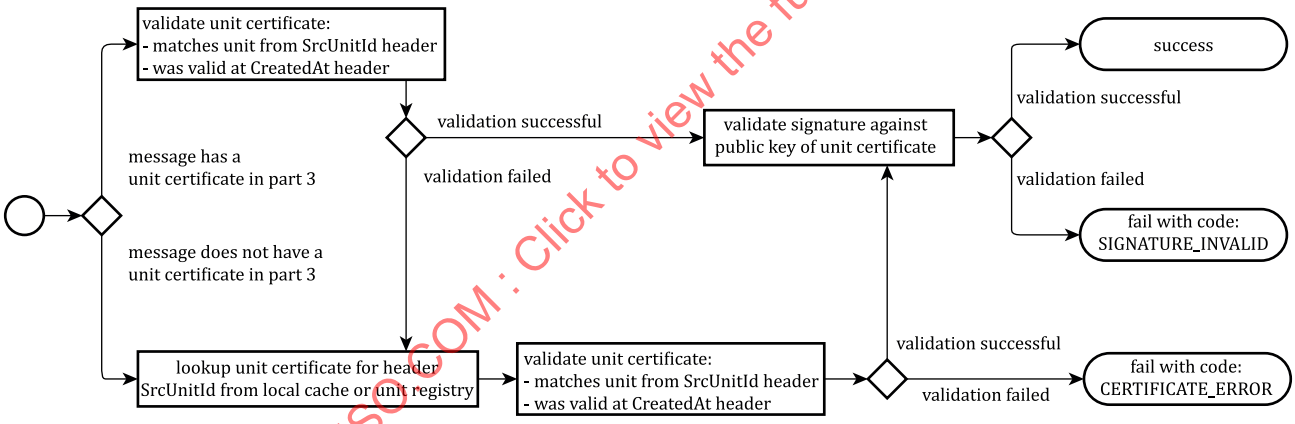


Figure 13 — Validating signature of signed message

6.10.6 Encrypting data

6.10.6.1 General

Whenever the data shall be secured against unauthorized reading, so that only the designated recipient of the data can read it, then encryption shall be used as follows:

a) the sender looks up the public key of the recipient (e.g. from a trusted certificate);

b) the data to secure is encrypted with that public key, so only the recipient can decrypt it;

c) the encrypted data are sent to the recipient;

d) the recipient uses the recipient's private key to decrypt the data;

e) the recipient checks that the decrypted data has an expected format to validate, and that the data has not been tampered with on the transport.

6.10.6.2 Message type “base.encrypted”

To encrypt one or multiple 4891-messages, they shall be wrapped into a message of `base.encrypted` type. This shall be done by following these steps:

a) lookup a trusted unit certificate for the destination unit;

b) build a list of the 4891-messages to be encrypted;

c) encode that list of messages into a JSON-encoded string;

d) encrypt that JSON-encoded string with the destination unit's public key from the unit certificate;

e) create a new message of type `base.encrypted` containing the encrypted data.

Table 30 specifies the composition for this message type.

Table 30 — Composition of 4891-message type “base.encrypted”

Message type	`base.encrypted`	
Data part 1	**Type**	`BINARY`
	Data	the encrypted messages

The receiving unit can only decrypt the contained messages if the unit has access to the private key matching the public key that has been used for encryption. The decryption process is done as follows:

f) decrypt the data from data part 1 Table 30 by using the private key;

g) validate that the decrypted data results in a JSON-encoded string;

h) decode the list of original 4891-messages from the decrypted JSON-encoded string.

6.11 Messaging

6.11.1 General

6.11.2 to 6.11.7 describe processes for handling and processing 4891-messages that shall be followed by implementations of 4891-units.

Additions to these processes are described in Annex A for integrating with controlled equipment.

6.11.2 Error message

To indicate failure of an operation triggered by another message, this message type shall be used. A more specific error code and potentially a description helping to debug shall be placed in the message’s data part.

Table 31 specifies the composition for this message type.

Table 31 — Composition of 4891-message type “base.error”

Message type	`base.error`				
Message header	**Field**		**Type**	**Required**	**Description**
	`ReferenceId`		`<uuid>`	yes	References the message that triggered this error.
Data part 1	**Encoding**	`RECORD`			
	Data	**Field**	**Type**	**Required**	**Description**
		`Code`	`<enum>`	yes	A technical short error code string. The expected values for this field shall be specified for operations that can fail.
		`Details`	`<string>`	no	Optional details describing the error, describing what went wrong to help debugging. If this field is present, English language shall be used for the content of the text in this field.

6.11.3 Message meta structure

All received 4891-messages shall be enriched with the following meta-information while they are processed.

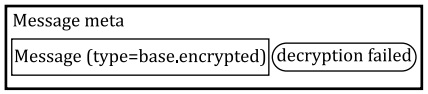
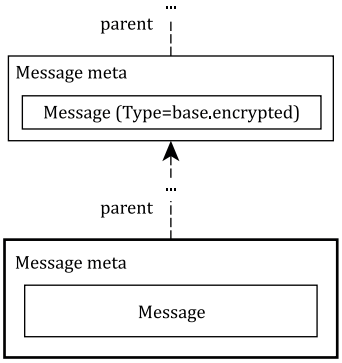
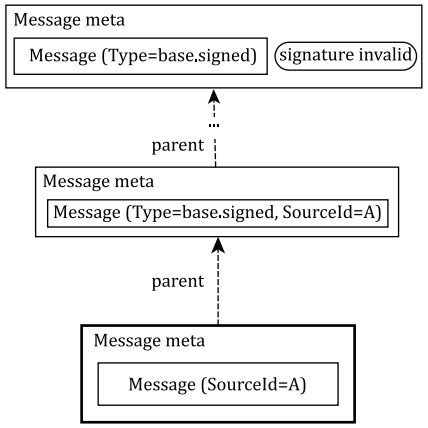
Table 32 specifies a structure for message meta-information that is referred to in the standardized processes illustrated in Figures 14 to 16. Implementations shall use an applicable representation, that can express at least the required information to implement the base processes described in 6.11.4 to 6.11.7.

Table 32 — Meta-information for a 4891-message that is being processed

Information	Type	Description
message	4891-message	The actual 4891-message.
parent message	reference to message meta structure	A reference to the meta structure of the 4891-message that this message was embedded in, i.e. a message of type `base.encrypted` or `base.signed`. Only set if this message was embedded in another message.
flag “decryption failed”	boolean	If set, this indicates that the message could not be decrypted, i.e. the message data are corrupt, or an unexpected key had been used for encrypting the message. Only set for messages of type `base.encrypted`.
flag “signature invalid”	boolean	If set, this indicates that the message's signature has failed verification, indicating that either the message's integrity has been broken (i.e. corrupted or tampered with) or that the signature could not be verified because the signee's public key could not be looked up. Only set for messages of type `base.signed`.

Table 33 gives examples for different value combinations of this structure.

Table 33 — Examples for meta-information structures of different 4891-messages

Example situation	Description
	This message has failed decryption, so the child messages embedded in it cannot be unpacked and processed.
	This message is embedded in other messages. Since at least one of the ancestor messages has type `base.encrypted`, it is considered to have been transported securely/encrypted. One of the ancestor messages had type `base.encrypted`, so this message can be handled as "has been transported securely/encrypted".
	This message is deeply embedded in a larger message list/tree. The direct parent message was created by the same unit and has properly signed this message. So far, the integrity can be considered intact. When looking further up in the ancestor list a message with broken signature can be found ("signature invalid"-flag). That indicates that some data integrity has been broken somewhere on the level between the message's parent and that broken signature message.

6.11.4 Receiving message from another unit

Figures 14 and 15 describe the processes of a 4891-unit receiving 4891-messages for further processing. All received messages shall be forwarded into the general message processing logic described in 6.11.5.

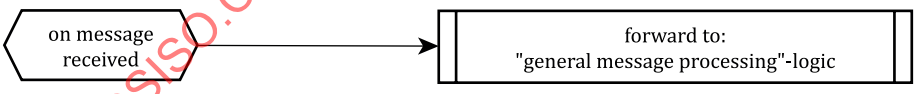


Figure 14 — Logic when receiving 4891-message from message broker

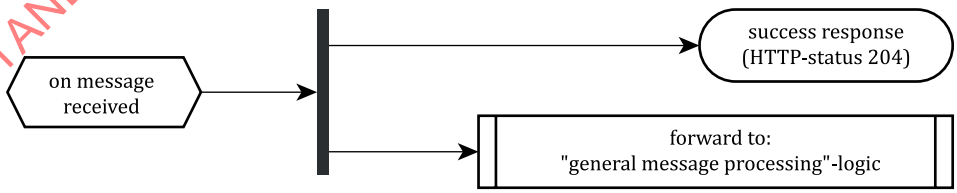


Figure 15 — Logic when receiving 4891-message via direct messaging API

6.11.5 General message processing logic

Figure 16 describes the general message processing logic that all 4891-units shall implement. This process standardizes the handling of incoming encrypted and signed messages across all 4891-units.

The process diagram contains two main parts:

a) A database to hold messages that shall be processed by the logic, i.e. messages that have been received from an external source or messages that have been unpacked from such messages for further processing.

b) A processing loop that waits for messages in that database and processes them one by one, potentially putting new messages into the database (i.e. when unpacking embedded messages from encrypted or signed message types).

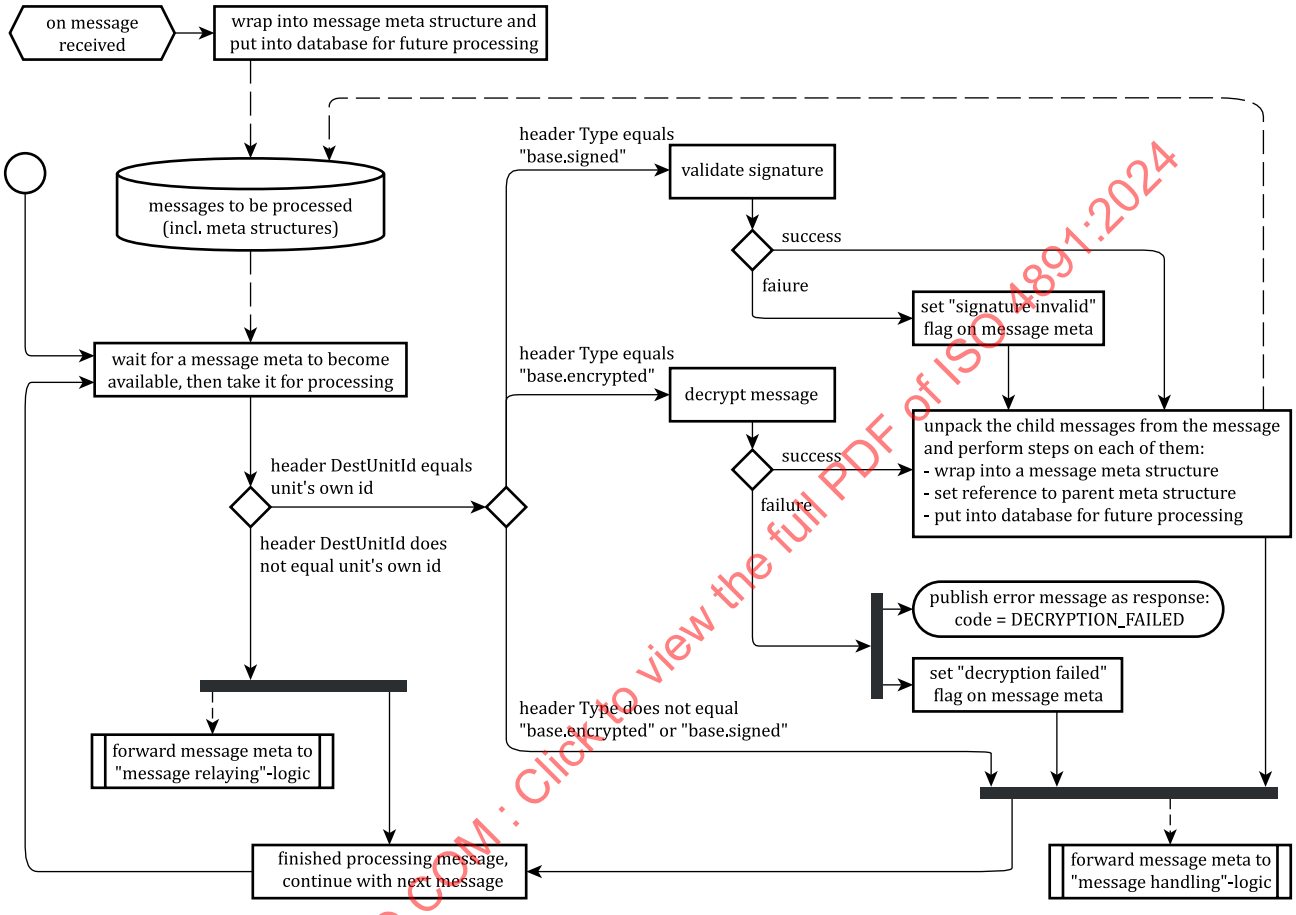


Figure 16 — General message processing logic

The application-specific use cases shall be implemented in the handling logic for handling or relaying a message that has passed through this process. The message handling and relaying processes shall be implemented as applicable. See 6.11.6 and 6.11.7 for further information on implementation of those.

6.11.6 Message relaying logic

The message relaying logic shall be implemented as applicable. A few further details on implementation are given in the following list for consideration:

— silently discard the message if relaying is not supported by implementation;

— relaying broadcasted messages is discouraged.

6.11.7 Message handling logic

The message handling logic shall be implemented as applicable. A few further details on implementation are given in the following list for consideration:

— the unit may publish messages as a response to certain handled messages;

— the handling logic will be invoked for each received message and for each of the messages embedded in them (i.e. for types of `base.signed` and `base.encrypted`)

Annex A provides a general message handling logic for 4891-smart gateway unit implementations and specifies requirements for these implementations.

7 Test methods

7.1 General

7.1.1 Manufacturable products

The test methods presented in 7.2 are structured in terms of manufacturable products, starting with the specific 4891-components that can be implemented on their own:

a) 4891-message broker

b) 4891-service discovery

c) 4891-unit registry

d) 4891-smart gateway unit

e) 4891-I/O unit

Manufacturers may choose to implement multiple 4891-components or a subset of those as dedicated products. Regardless of how many distinct 4891-components are part of the implementation, each of those shall be tested in isolation against the applicable clauses, being the equipment under test (EUT) in those clauses (see Table 34).

7.2 defines the test methods and setup requirements for each of the 4891-component types. Tests for shared common functionalities and behaviours are described in dedicated subclauses of 7.3, which are referenced in the clauses applicable to specific EUT. 7.1 defines general test requirements and guidelines on how on test.

All applicable test methods shall be fulfilled for EUT to be considered 4891-compliant. Additional test methods for 4891.A-compliant equipment can be found in A.11, and for 4891.B-compliant equipment in B.6. Consult Table 34 to identify the relevant clauses containing tests for EUT classes.

Table 34 — Overview of clauses containing tests applicable to different EUT classes

EUT class	Relevant clauses containing tests		
	4891-compliant (base standard)	**4891.A-compliant (with Annex A)**	**4891.B-compliant (with Annex A and B)**
4891-message broker	— 7.1 — 7.2.2 — 7.3.2 — additional subclauses of 7.3, as referenced	N/A	N/A
4891-service discovery	— 7.1 — 7.2.3 — 7.3.2 — additional subclauses of 7.3, as referenced	N/A	N/A
4891-unit registry	— 7.1 — 7.2.4 — 7.3.2 — additional subclauses of 7.3, as referenced		
4891-smart gateway unit	— 7.1 — 7.2.5 — 7.3.3 — 7.3.2 — additional subclauses of 7.3, as referenced	— 7.1 — 7.2.5 — 7.3.3 — 7.3.2 — A.11.1 — A.11.2.2 — additional subclauses of 7.3, as referenced	N/A
4891-I/O unit	— 7.1 — 7.2.6 — 7.3.3 — 7.3.2 — additional subclauses of 7.3, as referenced	— 7.1 — 7.2.6 — 7.3.3 — 7.3.2 — A.11.1 — A.11.2.3 — additional subclauses of 7.3, as referenced	N/A

Table 34 *(continued)*

EUT class	Relevant clauses containing tests		
	4891-compliant (base standard)	**4891.A-compliant (with Annex A)**	**4891.B-compliant (with Annex A and B)**
Controlled equipment (see Annex A and B)	N/A	— 7.1 — A.11.1 — A.11.2.1 — additional subclauses of 7.3, as referenced	— 7.1 — A.11.1 — A.11.2.1 — B.6.1 — B.6.2.1 — additional subclauses of 7.3, as referenced

7.1.2 Testing and classification

The requirements contained in this document specify the behaviour of software, and the manufactured products built according to this document are pieces of software. As software products are often updated and re-released frequently in new versions, it is not practical to perform a full classification cycle for each new release of the software.

Table 35 contains a proposal for a practical way of handling the testing and classification processes of products manufactured in accordance with this document.

Table 35 — Proposed steps for testing and classification processes

Situation	Proposed steps
While manufacturing the product	— the manufacturer creates and maintains test suites covering the relevant test methods — those tests may be a mixture of automatic software tests and manually applied tests
Initial classification of product	— the manufacturer creates a test report for the first version of EUT — the manufacturer applies the test suites against EUT — the manufacturer documents the test results in the report — the auditor verifies the content of the report — the auditor may specify a specific format or template to use for the test report — the auditor confirms that the EUT is conforming to this document — the manufacturer keeps test report for future reference

New release of product (new software version)	— the manufacturer creates a test report for the new version of EUT
	— the manufacturer keeps test report for future reference
	— the manufacturer identifies if the new release modifies any behaviour covered in requirements of this document, and notes down that fact in the test report.
	  — If this is the case:
	  — the manufacturer applies the full test suites against EUT;
	  — the manufacturer documents the results of those tests in the test report.
	  — If this is not the case:
	  — the manufacturer may apply and document the results of the full test suites against EUT, but is not required.

7.1.3 Use of simulated equipment

Certain tests require simulated equipment which provides the functionality used by EUT.

Testers shall implement simulated equipment that is required for testing the EUT in a way that allows inspection of communication behaviour and dataflows of the EUT with the simulated equipment, as specified in the test methods.

The simulated equipment shall cover the communication interfaces that the EUT is expected to interact with (e.g. HTTP-API servers, HTTP-API clients, MQTT-servers and MQTT-clients). When operating such simulated equipment for testing the EUT, the simulated equipment shall be instrumented (e.g. running from an IDE with active breakpoints) or log-files shall be created (e.g. incoming API requests and data), so that evidence can be collected to verify the behaviour and data-assertions stated in the test methods.

Depending on the test requirements, it is possible that the simulated equipment is also required to simulate certain internal logical behaviours of the 4891-component that is simulated (e.g. servers with persistent state, creation of certificates). In such cases, it can be convenient to utilize 4891-conforming equipment as “simulated equipment” if such equipment provides capabilities for instrumentation or logging as required to cover the test assertions for the EUT.

7.1.4 Testing of UDP broadcasts

For testing the broadcasting behaviour of UDP discovery packets, testers are advised to use network inspection tools, that allow monitoring of UDP traffic on the connected network(s), to observe the emission behaviour, origin, destination, and content of such packets. An example of such a tool is:

— Wireshark[2] available at https://www.wireshark.org/

7.1.5 Testing of HTTP-API servers

For testing HTTP-API server endpoints of the EUT, testers are advised to use HTTP-client tools, that allows fine control of HTTP-request headers, content-encoding, and inspection capabilities for received responses. Examples of such tools are:

— curl[3] available at https://curl.se/

— Postman[4] available at https://postman.com/

Testers shall study the EUT API server's expected behaviour for each specified API endpoint and prepare a test-suite to cover the standardized endpoint behaviour. The following requirements shall be considered to craft such test suite:

— for each endpoint (or combination of endpoints, if applicable), create sets of differently parametrized HTTP-requests, with parameters so that:
  — different combinations of query parameters are used, as applicable,
  — different values are used for query parameters to test valid value ranges, as applicable,
  — different values are used for content bodies to test different combinations of allowed content structure, as applicable,
  — success cases are covered, e.g. expecting EUT to trigger functionality, update with provided data or respond with requested data,
  — failure cases are covered, e.g. expecting error responses for wrong parametrization;
— verify that failure cases do not result in the EUT malfunctioning, e.g. not leading to non-recoverable error-states.

7.1.6 Testing of HTTP-API clients

For testing the request behaviour of HTTP-API clients of the EUT, testers are advised to use network inspection tools, that allow monitoring of HTTP traffic on the connected network(s), to observe emission behaviour, headers, origin, destination and content of such HTTP-requests. An example of such a tool is:

— Wireshark available at https://www.wireshark.org/

For testing complex HTTP-API interaction of the EUT, it is advised to operate already 4891-complying equipment for the EUT to interact with (see 7.1.2).

2) Whireshark is an example of a suitable product available commercially. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of this product.

3) curl is an example of a suitable product available commercially. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of this product.

4) Postman is an example of a suitable product available commercially. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of this product.

7.1.7 Inspecting 4891-messages exchanged between 4891-units

For inspecting streams of messages between two 4891-units, consider all messages on these channels:

— messages published on the 4891-message broker for a receiving unit;

— messages received on the direct messaging API server of the receiving unit.

If an MQTT-server is operated as part of the simulated message broker, it is advised to use an MQTT-client to inspect and monitor the message topics on that MQTT-server while interacting with the EUT. An example of such a tool is:

— MQTT Explorer[5] available at https://mqtt-explorer.com/

For messages transferred via direct messaging API, 7.1.5 and 7.1.6 apply.

Whenever a 4891-message shall be inspected, the steps of the "general message processing logic" (see 6.11.5) shall be followed to unpack potentially embedded messages, i.e.:

— for messages of type `base.signed`: extract the contained messages and inspect them;

— for messages of type `base.encrypted`: try to decrypt the contained messages with the private key of the receiving 4891-unit and inspect them.

That way, a single inspected `base.signed` or `base.encrypted` message can result in multiple messages (recursively unwrapped) to be inspected. If not stated otherwise, test methods refer to the embedded messages, e.g. the messages of types other than `base.signed` or `base.encrypted`.

7.2 4891-compliant equipment tests

7.2.1 General

7.2.2 to 7.2.6 specify the tests which are applicable for different types of manufacturable products (see Table 34).

7.2.2 4891-message broker tests

7.2.2.1 General

The tests described in 7.2.2 are applicable if the EUT implements a 4891-message broker.

Table 36 lists simulated equipment that is used in the tests of 7.2.2.

Table 36 — Simulated equipment for testing 4891-message broker on 4891-compliancy

Simulated equipment	Details
4891-component	— MQTT-client (as client A) for subscribing messages
4891-component	— MQTT-client (as client B) for subscribing messages
4891-component	— MQTT-client (as client C) for publishing messages

Perform the tests specified in 7.3.2 (see 5.2 and Figure 5).

5) MQTT Explorer is an example of a suitable product available commercially. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of this product.

7.2.2.2 Configuration tests

Perform the following tests:

a) Consult the manufacturer's documentation to determine how EUT can be configured. Verify that the access to configuration of EUT is protected against unauthorized access from other 4891-components, e.g. the configuration interface is not reachable from inside any network exposed to 4891-components, or it is secured by user credentials required to authenticate on the configuration interface (see 6.5).

b) Consult the manufacturer's documentation to determine if and how user authentication can be configured on the EUT. Perform those steps to configure user credentials or to look them up, if applicable (see 6.5.2).

7.2.2.3 MQTT-client connection tests

Use an MQTT-client to connect to the MQTT-server of EUT. Use the previously noted down user credentials if any have been configured [(see 7.2.2.2 b)].

Perform the following tests:

a) Verify that the configured user credentials can be used for connection. If no user credentials have been configured, verify that anonymous connections are possible (see 6.5.2).

b) Verify that protocol version `3.1.1` is supported by the server (see 6.5).

7.2.2.4 Messaging tests

Connect three different MQTT-clients (A, B and C, see Table 36) to the MQTT-server of EUT as described in 7.2.2.3.

On client A, subscribe to these MQTT-topics:

— `sappnet/msg/+/from/+/to/a`

— `sappnet/msg/+/from/+/to-all`

— `sappnet/msg/x.inspect/#`

On client B, subscribe to these MQTT-topics:

— `sappnet/msg/+/from/+/to/b`

— `sappnet/msg/+/from/+/to-all`

With client C, send any valid 4891-message on the topics listed in Table 37. For each of those, verify that the message is received or not received on clients A and B as stated in Table 37 (see 5.2.2).

Table 37 — List of MQTT-topics to send to a 4891-message for testing receipt

Topic no.	Topic on which client C publishes the message	Message is expected to be received	
		Client A	Client B
1	`unsupported.topic`	no	no
2	`sappnet/msg/x.anything/from/c/to/a`	yes	no
3	`sappnet/msg/x.anything/from/c/to/b`	no	yes
4	`sappnet/msg/x.anything/from/c/to/x`	no	no
5	`sappnet/msg/x.inspect/from/c/to/a`	yes	yes
6	`sappnet/msg/x.inspect/from/c/to/b`	yes	yes
7	`sappnet/msg/x.inspect/from/c/to/x`	yes	no
8	`sappnet/msg/x.broadcast/from/c/to-all`	yes	yes

7.2.3 4891-service discovery tests

7.2.3.1 General

The tests specified in 7.2.3.1 to 7.2.3.4 are applicable if the EUT implements a 4891-service discovery.

Table 38 lists simulated equipment that is used in the tests of 7.2.3.

Table 38 — Simulated equipment for testing 4891-service discovery on 4891-compliancy

Simulated equipment	Details
4891-component	— service discovery API client
4891-message broker	— MQTT-server

Perform the tests specified in 7.3.2 (see 5.2 and Figure 5).

7.2.3.2 Configuration tests

Perform the following tests:

a) Consult the manufacturer's documentation to determine how EUT can be configured. Verify that the access to configuration of EUT is protected against unauthorized access from other 4891-components, e.g. the configuration interface is not reachable from inside any network exposed to 4891-components, or it is secured by user credentials required to authenticate on the configuration interface (see 6.6).

b) Consult the manufacturer's documentation to determine how service connectors can be configured on EUT (see 5.2.3). Follow the documented steps and verify the following:

— The configuration is secured in some way, so that no unauthorized user can perform the configuration (see 5.2.3 and 6.6.2).

— The connector for 4891-message broker can be configured (see 5.2.3, 6.5.2 and 6.6.2.2).

— The connector for 4891-unit registry can be configured (see 5.2.3, 6.6.2.3, and 6.7).

— The connector for 4891-smart gateway unit can be configured (see 5.2.3, and 6.6.2.4)

— The configuration of connector for 4891-smart gateway unit is optional (see 5.2)

7.2.3.3 Service discovery API discovery packet tests

Perform the following tests:

a) Apply all tests from 7.3.12 (with the parameters specified in 6.6.5).

b) Verify by observation, that the API discovery packet is broadcast periodically every 5 s to 10 s (see 6.6.5).

7.2.3.4 Service discovery API server tests

Extract the service discovery API base URL from the discovery UDP packets broadcast by the EUT (see 7.2.3.3). Use that base URL to craft the HTTP-endpoint URLs for the tests in this subclause (see 6.6.5).

Perform the following tests:

— all tests specified in 7.3.14 (see 6.6.4).

— Create a suite of endpoint specific test cases according to 7.1.5. With those, verify that all endpoints behave as specified in this document. Use Table 39 as a guideline for some specific tests to perform, additionally to the general tests proposed in 7.1.5.

Table 39 — Proposed tests for endpoints of 4891-service discovery API server

Endpoint	Proposed tests
GetServiceConnectors (see 6.6.4.2 and 5.2.3)	— verify that the response contains all the connectors configured on the EUT — change the configured connectors on the EUT and verify that the new connectors are reflected in the response

7.2.4 4891-unit registry tests

7.2.4.1 General

The tests specified in 7.2.4.2 to 7.2.4.4 are applicable if the EUT implements a 4891-unit registry.

Table 40 lists simulated equipment that is used in the tests of 7.2.4.

Table 40 — Simulated equipment for testing 4891-unit registry on 4891-compliancy

Simulated equipment	Details
4891-service discovery (maybe)	— service discovery API server, incl. UDP broadcasting
4891-unit	— unit registry API client
4891-message broker	— MQTT-server

Perform the tests specified in 7.3.2 (see 5.2 and Figure 5).

7.2.4.2 Configuration tests

Perform the following tests:

a) Consult the manufacturer's documentation to determine how EUT can be configured. Verify that the access to configuration of the EUT is protected against unauthorized access from other 4891-components, e.g. the configuration interface is not reachable from inside any network exposed to 4891-components, or it is secured by user credentials required to authenticate on the configuration interface (see 6.7).

b) Verify by observation, that the EUT provides functionality to configure root certificates to be used for signing unit certificates issued by the EUT (see 5.2.4 and 6.10.2). Verify that external root certificates can be imported, or that the EUT can create self-signed root certificates (see 6.10.3.2). The manufacturer's documentation shall be consulted to determine which is applicable (see 6.10.2).

c) If external certificates can be imported as root certificates, verify that EUT only accepts certificates that pass all tests from 7.3.9, i.e. that EUT rejects non-passing certificates (see 6.10.3).

d) If the EUT allows the creation of self-signed root certificates, verify that such certificates pass all tests from 7.3.9 (see 6.10.3.2).

If only one root certificate can be configured:

e) consider it to be the main root certificate for all tests in 7.2.4.

If multiple root certificates can be configured:

f) verify by observation, that always exactly one of those configured certificates is flagged as the main root certificate (see 6.10.3.3);

g) verify by observation, that the selection of the main root certificate is either done automatically, or that it can be configured manually (see 6.10.3.3).

If the selection of the main root certificate is done automatically be the EUT:

h) verify by observation, that the selection algorithm either follows the proposed algorithm, or that the algorithm implemented by EUT is described in the manufacturer's documentation (see 6.10.3.3).

7.2.4.3 Unit registry API server tests

Lookup or configure the base URL for the unit registry API server of EUT. Consult the manufacturer's documentation to determine how that can be accomplished. Use that base URL to craft the HTTP-endpoint URLs for the following tests in this clause (see 6.7.4).

Perform the following tests:

a) All tests specified in 7.3.14 (see 6.7.4).

b) For endpoints referring root certificates, apply all tests from 7.3.9 on those certificates.

c) For endpoints referring unit certificates, apply all tests from 7.3.10 on those certificates.

d) Create a suite of endpoint specific test cases in accordance with 7.1.5. With those, verify that all endpoints behave as specified in this document. Use Table 41 as a guideline for some specific tests to perform, additionally to the general tests proposed in 7.1.5.

Table 41 — Proposed tests for endpoints of 4891-unit registry API server

Endpoint	Proposed tests
`GetActiveRootCertificates` (see 5.2.4 and 6.7.4.2)	— verify that the response lists all active root certificates that are configured — remove or replace active root certificates, verify that those are not listed in the response anymore (see 6.10.3.7)
`RegisterUnit` (see 5.2.4 and 6.7.4.3)	— register with a new key-pair, verify a new unit identifier is returned — register again with same key-pair, verify that the same unit identifier is returned
`ModifyUnitInformation` (see 6.7.4.4)	— verify that the endpoint is protected with unit authentication (see 7.3.14.2)
`GetUnitInformation` (see 5.2.4, and 6.7.4.5)	— request with non-existing unit identifier, verify error status 404 — request with unit identifier from previous registration, verify response contains that unit's information and unit certificate
`GetListOfUnitInformation` (see 5.2.4, and 6.7.4.6)	— before registering any units, verify that response is an empty list — register a few units, verify that those show up in the response list — use different combinations of filters, sorting and pagination query parameters, verify that the response list is respecting those parameters

7.2.4.4 Unit registration tests

The tests in this subclause use endpoints of the unit registry API server operated by the EUT. Prepare the EUT with necessary configuration, so that it is ready to serve requests.

For each of the tests in this subclause, whenever a unit certificate is returned in the response data, verify that the unit certificate passes all tests listed in 7.3.10, with the unit identifier that is received in the same response.

Perform the following tests:

a) Before any root certificates are configured on the EUT: generate a new key-pair and use the endpoint `RegisterUnit` of the unit registry API to register a new 4891-unit on the EUT. Verify by inspection, that the response indicates an error because of no active root certificate being configured (see 6.10.4.2).

b) Configure a root certificate on the EUT as instructed in the manufacturer's documentation. Select that root certificate as main root certificate, if applicable.

c) Generate a new key-pair and use the endpoint `RegisterUnit` of the unit registry API to register a new 4891-unit on the EUT. Verify by inspection of the response, that the returned unit certificate is signed

by the main root certificate, configured in previous step (see 6.10.4.2). Note down the received unit identifier.

d) Configure a new root certificate on EUT. Select that new certificate as main root certificate, if not done automatically.

e) Use the endpoint `GetUnitInformation` with the previously noted down unit identifier from step c). Verify by inspection of the response data, that the unit certificate eventually changed, it being signed by the new main root certificate. Repeat this test if the certificate did not change immediately. Consult the manufacturer's documentation to determine how long it takes for unit certificates to be renewed with a new main root certificate (see 6.10.4.2).

f) Use the `RegisterUnit` endpoint again, this time with a new key-pair. Verify by inspection of the response data, that this results in a new unit being registered, i.e. a new unique unit identifier is returned with a new unit certificate for it (see 6.10.4.2).

g) Use the `RegisterUnit` endpoint again, this time with the same public key as the test in f). Verify by inspection of the response data, that the same unit identifier and unit certificate are returned as in the previous test (see 6.10.4.2).

h) Use the endpoint `GetUnitInformation` for each unit identifier received in tests a) to g). Verify by inspection of the response data, that each of those returned information and unit certificates match the used unit identifier (see 6.7.2, see 6.10.4.2).

i) Use the endpoint `GetUnitInformation` with a random new UUID value. Verify by inspection of the response data, that it indicates an error, because of no such unit being found.

j) Use the endpoint `GetListOfUnitInformation`. Verify by inspection of the response, that the returned list contains the information and unit certificates for all the units registered in tests a) to g).

7.2.4.5 Unit tracking tests

Some tests in this subclause use endpoints of the unit registry API server operated by the EUT. Prepare the EUT with necessary configuration, so that it is ready to serve requests.

Some tests in this clause require interaction with the 4891-message broker. Operate a simulated 4891-message broker for those tests.

Perform the following tests:

a) Generate a few key-pairs and use the endpoint `RegisterUnit` for each of those to register a few 4891-units. For each note down the time when the request was performed, and the unit identifier returned for it.

b) Use the endpoint `GetUnitInformation` for each of the received unit identifiers. Verify by inspection of the response data, that the timestamps of registration and last activity approximately match the noted down timing from the registration request. Verify that both timestamps (registration and last activity) are equal (see 6.7.2).

c) Use the endpoint `ModifyUnitInformation` for some of the unit identifiers, changing their names. Use the same endpoint for the other unit identifiers, but this time perform an invalid request (i.e. providing wrong authentication information).

d) Verify by inspection of the response, that the “last activity” timestamps changed for all the units, which the previous `ModifyUnitInformation` requests succeeded. Verify that this did not happen for the other unit identifiers (see 6.7.2.2).

e) Use all the endpoints other than `ModifyUnitInformation` with valid parametrization. Verify by requesting the units' information afterwards, that none of the “last activity” timestamps of previously registered units were changed by that (see 6.7.2.2).

f) Create and publish the 4891-messages listed in Table 42 on the simulated 4891-message broker. After each of those messages, use the `GetListOfUnitInformation` endpoint to verify, that the expected "last activity" timestamps did update, according to Table 42 (see 6.7.2.3). Different units (all registered in previous steps) are differentiated by letters as placeholders for their specific unit identifiers (e.g. A, B ...).

Table 42 — Proposed 4891-messages for testing unit tracking behaviour of 4891-unit registry

Message no.	Headers of of outer message (type `base.signed`)		Headers of inner message (type `x.anything`)		Expected changes of "last activity" timestamps on unit information
	`SrcUnitId`	`CreatedAt`	`SrcUnitId`	`CreatedAt`	
1	-		A	before A's "last activity"	A's last activity not updated
2	-		A	after A's "last activity"	A's last activity = inner `CreatedAt`
3	B	before B's "last activity"	A	before A's "last activity"	A's last activity not updated B's last activity not updated
4	B	before B's "last activity"	A	after A's "last activity"	A's last activity = inner `CreatedAt` B's last activity not updated
5	B	after B's "last activity"	A	before A's "last activity"	A's last activity not updated B's last activity = outer `CreatedAt`
6	B	after B's "last activity"	A	after A's "last activity"	A's last activity = inner `CreatedAt` B's last activity = outer `CreatedAt`

7.2.5 4891-smart gateway unit tests

7.2.5.1 General

The tests specified in 7.2.5.2 are applicable if the EUT implements a 4891-smart gateway unit.

Table 43 lists simulated equipment that is used in the tests specified in 7.2.5.2.

Table 43 — Simulated equipment for testing 4891-smart gateway unit on 4891-compliancy

Simulated equipment	Details
4891-service discovery	— service discovery API server, incl. UDP broadcasting
4891-unit registry	— unit registry API server
4891-message broker	— MQTT-server
4891-unit (maybe)	— direct messaging API server
4891-unit (maybe)	— direct messaging API client

Perform the following tests:

a) All tests specified in 7.3.3 (see 5.2 and Figure 5).

b) Verify by inspection of manufacturer's documentation, that the EUT can be installed onboard of a vessel (e.g. it is pre-installed on a physical device, or can be installed on an existing server onboard), so that it can be connected to the local network (see 5.2.5.2).

7.2.5.2 4891-message broker subscriptions tests

Perform the following tests:

a) Verify by inspection that the EUT is running an MQTT-client which is connected to the simulated 4891-message broker. This shall be done by looking through the active MQTT-connections on the MQTT-server of the simulated message broker (see 5.2.5.2).

b) Send an arbitrary 4891-message content to each MQTT-topic specified in Table 44. Replace the placeholders in those MQTT-topics according to Table 45. Verify that the MQTT-client of EUT has subscribed to at least the topics marked with "transmission expected" in Table 44 (see 5.2.5.2). This

shall be done by inspecting the network traffic between the MQTT-server of the simulated message broker and the MQTT-client of the EUT.

Table 44 — MQTT-topics for testing message subscribing behaviour of 4891-smart gateway unit

Topic no.	MQTT-topic	Transmission expected
1	"sappnet/msg/<any-type>/from/<random-uuid>/to-all"	yes
2	"sappnet/msg/<any-type>/from/<random-uuid>/to/<smart-gateway-unit-id>"	yes
3	"sappnet/msg/<any-type>/from/<random-uuid>/to/<random-uuid>"	no

Table 45 — Placeholders for MQTT-topics in tests of 4891-smart gateway unit

Placeholder	Content
<any-type>	A random value denoting some 4891-message type. Examples include: — "base.signed" — "x.some.custom" — "future.standard.type"
<random-uuid>	A random UUID, that is not the unit identifier of the EUT. Examples include: — "71619643-eb71-46b9-bf25-37ee120ded9c" — "10b0bb31-af80-4ad1-92a4-481f4384d0e8"
<smart-gateway-unit-id>	The unit identifier of the EUT. Examples include: — "71619643-eb71-46b9-bf25-37ee120ded9c" — "10b0bb31-af80-4ad1-92a4-481f4384d0e8"

7.2.6 4891-I/O unit tests

7.2.6.1 General

The tests of 7.2.6 are applicable if the EUT implements a 4891-I/O unit.

Table 46 lists simulated equipment that is used in the tests of 7.2.6.

Table 46 — Simulated equipment for testing 4891-I/O unit on 4891-compliancy

Simulated equipment	Details
4891-service discovery	— service discovery API server, incl. UDP broadcasting
4891-unit registry	— unit registry API server
4891-message broker	— MQTT-server
4891-unit (maybe)	— direct messaging API server
4891-unit (maybe)	— direct messaging API client

Perform all tests specified in 7.3.3 (see 5.2 and Figure 5).

7.3 Shared functionality tests

7.3.1 General

7.3 contains tests for functionalities and behaviours that are shared by different types of manufacturable products. Some specific information for EUT is also included.

7.3.2 4891-component tests

7.3.2.1 General

Perform the following tests:

a) Verify that the manufacturer's documentation defines the type of component that the EUT implements, i.e. 4891-message broker, 4891-service discovery, 4891-unit registry, 4891-smart gateway unit, or 4891-I/O unit (see 6.1).

b) Apply applicable tests from 7.3.3 if the EUT implements a 4891-unit.

c) Apply applicable tests from 7.3.4 if the EUT interacts with the 4891-message broker.

d) Apply applicable tests from 7.3.5 if the EUT interacts with the 4891-service discovery API server.

e) Apply applicable tests from 7.3.6 if the EUT interacts with the 4891-unit registry API server.

f) Apply applicable tests from 7.3.8 if the EUT provides functionality to relay 4891-messages that it received.

7.3.2.2 Trusted communication tests

The tests specified in this subclause are applicable if the EUT is supporting trusted communication with other 4891-units. Consult the manufacturer's documentation to determine if that is the case for the EUT.

Consult the manufacturer's documentation to determine if EUT produces 4891-messages of type `base.encrypted` and how the functionalities producing them can be triggered (see 5.7 and 6.10.2). Trigger any such functionality with a registered but simulated 4891-unit (i.e. having access to unit's key-pair) as the destination. Perform the following tests on messages of type `base.encrypted` produced that way (see 5.6):

a) Verify that the message is structured as specified according to the type.

b) Verify that the message contains an encrypted message, that can be decrypted with the simulated unit's private key.

c) Verify that the decrypted message has header `DestUnitId` as the simulated unit's identifier.

d) Modify the encrypted message binary in any way. Verify that decrypting that modified binary is not resulting in a valid message structure anymore.

Consult the manufacturer's documentation to determine if EUT produces 4891-messages of type `base.signed` and how the functionalities producing them can be triggered (see 5.7 and 6.10.2). Trigger any such functionality. Perform the following tests on messages of type `base.signed` produced that way (see 5.6):

e) Verify that the message is structured as specified according to the type.

f) Verify that the message contains another message encoded as string.

g) Verify that the message contains a signature binary encoded as Base64-string.

h) Lookup the EUT's public key from the simulated 4891-unit registry. Verify that the signature from the message is valid for the looked up public key and string encoded contained message.

i) Modify the contained message string in any way. Verify that the signature is not validating anymore.

Consult the manufacturer's documentation to determine if any functionality of the EUT requires inbound 4891-messages to be signed, i.e. messages expected to be wrapped in 4891-messages of type `base.signed` (see 6.10.2). For any such functionality, perform the tests listed in Table 47 according to the applicable scenario.

Table 47 — Tests for message signing in applicable 4891-component functionality

Test no.	Test scenario	Test
1	Create a message that the EUT should accept on related functionality and send it to the EUT without signing it.	Verify that EUT is not triggering related functionality (missing signature).
2	Create a similar message, but this time wrap it in a message of type `base.signed` by using the key-pair of a unit whose signature the EUT should not accept for the functionality.	Verify that the EUT is not triggering related functionality (non-acceptable unit).
3	Create a similar message, but this time wrap it in a message of type `base.signed` by using the key-pair of a unit whose signature the EUT should accept for the functionality.	Verify that the EUT is not triggering the related functionality (invalid signature).
4	Create a similar message, but this time wrap it in a message of type `base.signed` by using the key-pair of a unit whose signature the EUT should accept for the functionality.	Verify that the EUT is triggering the related functionality.

Consult the manufacturer's documentation to determine if any functionality of the EUT requires inbound 4891-messages to be encrypted, i.e. messages expected to be wrapped in 4891-messages of type `base.encrypted`. For any such functionality, perform the tests in Table 48 according to the applicable scenario.

Table 48 — Tests for message encryption in applicable 4891-component functionality

Test no.	Test scenario	Test
1	Create a message that EUT should accept on related functionality and send it to the EUT without encrypting it.	Verify that EUT is not triggering related functionality. (not encrypted)
2	Create a similar message, but this time wrap it in a message of type `base.encrypted` by using any public key that is not registered for EUT.	Verify that the EUT is not triggering related functionality. (decryption failed)
3	Create a similar message, but this time wrap it in a message of type `base.encrypted` by using the public key registered for EUT. Afterwards, modify the encrypted binary data before sending it to EUT.	Verify that the EUT is not triggering the related functionality. (decryption failed)
4	Create a similar message, but this time wrap it in a message of type `base.encrypted` by using the public key registered for EUT.	Verify that the EUT is triggering the related functionality.

7.3.2.3 Root certificates trust tests

The tests specified in a) to c) below are applicable if the EUT fetches root certificates from connected 4891-unit registry. To determine this, connect the EUT with a simulated 4891-unit registry and observe if EUT fetches the root certificates via unit registry API.

Peform the following tests:

a) Verify that the EUT fetches root certificates from the unit registry API (see 6.10.3.4). To do so, connect the EUT to a simulated 4891-unit registry and observe that the `GetActiveRootCertificates` endpoint is invoked (see test methods in 7.3.6).

b) Verify by observation that EUT provides functionality to review root certificates, that it has received over time (see 6.10.3.6). Consult the manufacturer's documentation to determine how that functionality is accessed and used.

c) Verify by observation that EUT provides functionality to flag received root certificates as either trusted or untrusted (see 6.10.3.6). Consult the manufacturer's documentation to determine how that functionality is accessed and used.

7.3.2.4 Certificate validation tests

Tests specified in Table 49 are applicable if EUT performs functionality that requires root certificate validation. Consult the manufacturer's documentation to determine if any such functionality exists. Examples of such functionalities include:

— validating signatures produced from unit certificates;

— encrypting data with a unit's public key.

Functionality matching those examples is called trigger functionality in this clause.

Before the triggering functionality is performed, it is expected that the EUT performs these steps:

— validating the unit certificate used for the functionality (see 6.10.4.6);

— validating the root certificate used for signing the unit certificate (see 6.10.3.7).

Perform tests according to the Table 49 to verify the expected behaviour of the EUT. In those test scenarios, perform the trigger functionality as instructed.

Table 49 — Tests for certificate validation in applicable 4891-component functionality

Test no.	Test scenario	Test
1	a) On simulated 4891-unit registry, configure root certificate A. b) Wait until EUT requests that root certificate. c) Flag root certificate A as trusted on EUT. d) Prepare condition for the trigger functionality utilizing root certificate A. e) Remove root certificate A from unit registry again (i.e. revoke it). f) Wait until EUT requests root certificates again from the unit registry. Consult the manufacturer's documentation to determine if that behaviour can be enforced somehow. g) Try to perform the trigger functionality.	Verify that the trigger functionality can no longer be performed or that doing so results in an error. (non-active certificate not valid)
2	Same as test no. 1, except: — skip step b) and c) (EUT does never receive root certificate A)	Verify that the trigger functionality cannot be performed anymore or that doing so results in an error. (non-active certificate not valid)
3	Same as test no. 1, except: — skip steps e) and f) (revocation) — insert before step g): flag root certificate A as untrusted on EUT	Verify that the trigger functionality can no longer be performed or that doing so results in an error. (explicitly flagged as untrusted)
4	Same as test no. 1, except: — skip step c) (flag as trusted) — skip steps e) and f) (revocation)	Verify that the trigger functionality can't be performed anymore or that doing so results in an error. (untrusted by default)
5	Same as test no. 1, except: — skip steps e) and f) (revocation)	Verify that the trigger functionality can be performed as expected.

7.3.3 4891-unit tests

7.3.3.1 General

Peform the following tests:

a) those specified in 7.3.2 (see 5.2 and Figure 5).

b) those specified in 7.3.6 (see 6.7.3 and 6.8).

c) those specified in 7.3.4 (see 6.8).

d) those specified in 7.3.4.4 with topics related to the EUT as a 4891-unit (see 6.8).

e) those specified in 7.3.4.3 if the EUT publishes 4891-messages on the shared 4891-message broker (see 6.8). Consult the manufacturer's documentation to determine if that is the case.

f) Consult the manufacturer's documentation to determine if the EUT supports the client-side of interacting with other 4891-units via direct messaging API (i.e. looking up remote unit's meta-data or sending 4891-messages directly to it). If supported, then apply all tests from 7.3.7 (see 5.5 and 6.8).

g) Consult the manufacturer's documentation to determine if the EUT supports the server-side of interacting with other 4891-units via direct messaging API (i.e. reacting to lookup requests of own meta-data or accepting 4891-messages directly from a remote unit). If supported, then apply all tests from 7.3.3.5 (see 5.5 and 6.8).

h) Consult the manufacturer's documentation to determine if the EUT supports non-standard message types. Verify that those non-standard message types are documented together with the expected message structure and usage semantics. Verify by inspection of all 4891-messages emitted by the EUT, while performing any of the tests against this document, that all produced messages either have a standard message type or have a non-standard message type that is in the previously checked documentation (see 5.3.5).

7.3.3.2 Unit registration process tests

Connect the EUT into the simulated 4891-environment.

Consult the manufacturer's documentation to determine if the EUT registers itself as a 4891-unit automatically or if any manual steps are required to initiate that process. Perform the steps necessary to initiate the unit registration process.

Perform the following tests:

a) Verify by inspection of network communication (see 6.7), that the following events happen:

   — EUT fetches the connectors from the 4891-service discovery API server;

   — EUT fetches the active root certificates from the 4891-unit registry API server;

   — EUT tries to register itself as a 4891-unit on the 4891-unit registry API server (see 6.7, and 6.10.4.2) by sending its unique public key to the EUT, receiving a unique unit identifier and unit certificate as response.

b) Repeat the unit registration process and each time vary the response behaviour of the simulated 4891-unit registry to test the error handling of the EUT. Verify by observation that all tests listed in Table 50 are handled correctly by EUT.

Table 50 — Tests for unit registration process of 4891-unit

Test no.	Simulated behaviour of 4891-unit registry API	Expected behaviour of EUT
1	Return an error response on registration request.	EUT tries to register itself again after some delay automatically, or after a new attempt is triggered explicitly by some interaction.
2	Return a unit certificate that is not signed by any of the active root certificates.	EUT does detect that its received unit certificate is not valid (root certificate used for issuing it is either new or revoked). EUT mitigates this situation by eventually requesting an updated list of active root certificates from the 4891-unit registry. If the used root certificate is still not showing up in that list, EUT will eventually request its own user information again from unit registry, either by trying to register again with its public key (same as before) or by requesting on the endpoint `GetUnitInformation`.
3	Return a unit certificate that does not match the returned unit identifier.	EUT behaves as if an error response was returned (see test no. 1).

7.3.3.3 4891-message creation tests

The tests in this subclause are applicable to all 4891-messages created by EUT. Consult the manufacturer's documentation to determine which functionalities create 4891-messages (see 5.2.5). Ignore any functionality that is only relaying 4891-messages, that EUT received from other equipment.

Perform the following tests:

a) Trigger any such identified functionality of EUT that should lead to emission of 4891-messages by the EUT. For all 4891-messages emitted by EUT, verify by inspection that those messages have the following properties:

   — `Id` header is set to unit identifier of EUT;

   — that all tests from 7.3.11 have been applied.

b) Consult the manufacturer's documentation to determine if EUT emits signed 4891-messages, e.g. if it creates messages of type `base.signed` by utilizing its own private key. If that is the case, trigger any functionality that emits such a message and verify by inspection that the emitted message has the following properties (see 6.10.5):

   — the message's structure and content follow requirements specified in 6.10.5;

   — the signature in data part 2 validates for the public key of the EUT (lookup from 4891-unit registry by EUT's unit identifier) against the content of data part 1;

   — if the message has data part 3, then it contains the unit certificate of EUT that matches the unit certificate that can be requested from 4891-unit registry for the EUT's unit identifier.

c) Consult the manufacturer's documentation to determine if EUT emits encrypted 4891-messages only to be readable by the destination 4891-unit, e.g. if it creates messages of type `base.encrypted` by utilizing the public key of that destination unit. If that is the case, trigger any functionality that emits such a message with destination being a simulated 4891-unit, and verify by inspection, that the emitted message has the following properties (see 6.10.6):

   — the message's structure and content follow requirements specified in 6.10.6;

   — the binary content in data part 1 can be decrypted by using the private key of the simulated destination 4891-unit;

   — the decrypted content of data part 1 is a valid JSON encoded 4891-message itself.

d) Consult the manufacturer's documentation to determine if EUT provides functionalities that can emit 4891-messages of type `base.error`, e.g. a functionality that can respond with an error response message to a triggered functionality. Trigger any such functionality in a way, that should produce such error message and verify by inspection, that the emitted error message has the following properties (see 6.11.2):

   — the message's content follows requirements specified in 6.11.2;

   — the `ReferenceId` header contains the `Id` value of the message that was sent to the EUT for triggering the failed operation;

   — the record in data part 1 contains a code that is describing the error situation or a generic problem;

   — if the record in data part 1 contains a `Details` field, that field is filled with some details or a description of the error that occurred.

7.3.3.4 4891-message handling tests

Consult the manufacturer's documentation to determine which types of 4891-messages are handled by the EUT and what type of functionality is triggered by those messages. Additionally, lookup all message headers used in those messages.

Perform the following tests:

a) Verify for all those message types and header names, that they are well-formed according to “standard vs. non-standard” naming requirements (see 5.3.3 and 5.3.5). Select any of the identified message types as the test message type.

Select any of the identified message types as the "test message type" for tests b) to c).

The following list defines the properties of a baseline message, which is referenced in the test cases in Table 51:

— set header `Type` to any applicable value identified in manufacturer's documentation, not being “`base.signed`”, “`base.encrypted`”, or “`base.error`”;

— set header `Id` to a random UUID;

— set header `CreatedAt` to time of sending;

— set header `SrcUnitId` to any UUID other than EUT's unit identifier;

— set header `DestUnitId` to EUT's unit identifier;

— set additional headers and message parts according to assigned `Type`.

b) Apply the tests in Table 51 to cover the “general message processing”-logic for incoming 4891-messages (see 6.11.5). For each test-case, create a 4891-message as described below as a baseline. Some tests describe modifications to the message to be performed for that specific test. Send the resulting message to the EUT via a simulated 4891-message broker. If the EUT supports direct messaging functionality, repeat those tests, but this time send the messages to the direct messaging API server of the EUT.

c) Perform the tests specified in Table 51 again, but this time set the `DestUnitId` header on the outer most message that is sent to EUT to a unit identifier of a registered simulated 4891-unit that is not the EUR. Verify that EUT behaves in the following way for each test:

   — EUT is not handling any of the messages, as they have a different destination. This expected behaviour includes all the messages that can be embedded in the outer most messages, e.g. tests utilizing signing and encrypting messages (see 6.11.5).

   — If EUT supports message relaying, the messages are eventually emitted onto the 4891-message broker or forwarded to the destination 4891-unit in other ways. Consult the manufacturer's

documentation to determine if EUT supports this functionality and how it should behave. See also 7.3.8 for additional applicable tests of that functionality.

Table 51 — Tests for "general message processing" logic of 4891-unit

Test no.	4891-message properties	Expected effects of test
1	— use baseline message	The message is handled on EUT, producing the effects specified for the message's type in a standard or in the manufacturer's documentation (see 6.11.7).
2	— start with baseline message — set `Type` to a value that is not a message type supported by EUT	The message is either ignored by EUT, or EUT emits an error response message, indicating that the message is not supported on EUT (see 6.11.7).
3	— use baseline message as "inner message" — register a simulated 4891-unit on the 4891-unit registry ("signee unit") — use the private key of the signee unit to wrap the message in a message of type `base.signed` in the proper way (see 6.10.5)	The nested inner message is handled on EUT, producing the effects specified for the message's type in a standard or in manufacturer's documentation (see 6.11.7).
4	— create a message as in test no. 3 — sign the message again with the same or another registered simulated 4891-unit — repeat previous step a few times, effectively increasing the nesting depth of the inner message intended to be handled by EUT	The deeply nested inner message is handled on EUT, producing the effects specified for the message's type in a standard or in manufacturer's documentation (see 6.11.7).
5	— set `Type` to `base.encrypted` — add a data part and fill it with random binary data (Base64-encoded)	EUT emits an error response message with error code `DECRYPTION_FAILED` (see 6.11.5). Verify that this message references the original message via `ReferenceId`.
6	— use baseline message as "inner message" — use the public key of EUT (lookup in 4891-unit registry) to wrap the message in a message of type `base.encrypted` in the proper way (see 6.10.6)	The nested inner message is handled on EUT, producing the effects specified for the message's type in a standard or in manufacturer's documentation (see 6.11.7).
7	— create a message as in test no. 6 — encrypt the message again with EUT's public key — repeat previous step a few times, effectively increasing the nesting depth of the inner message intended to be handled by EUT	The deeply nested inner message is handled on EUT, producing the effects specified for the message's type in a standard or in manufacturer's documentation (see 6.11.7).

7.3.3.5 Direct messaging API server tests

The tests of this clause are applicable if the EUT provides functionality which allows other 4891-units connect to the EUT via direct messaging API, i.e. if EUT is operating a direct messaging API server.

Perform the following tests:

a) Apply all tests from 7.3.12 with the parameters specified in 6.9.4.

b) Verify by inspection that EUT is not broadcasting the API discovery packet in the following situations:

— before the EUT is registered as 4891-unit, i.e. before it has received a unit identifier and unit certificate from the 4891-unit registry (see 6.9.4);

— after the EUT has deleted its 4891-unit identity (key-pair, unit identifier), e.g. after a factory reset.

c) Verify that the API server is reachable in the following situations:

— whenever the API discovery packet is broadcast (see 6.9.4).

d) Verify by observation, that the API discovery packet is broadcast periodically (see 6.9.4):

— every 5 s to 60 s, if the API server is running permanently;

— more frequently, if the API server is running for short time frames only, i.e. if enabled for a short data transmission session with another 4891-unit.

Extract the base URL of the direct messaging API server from the discovery UDP packets broadcast by the EUT. Use that base URL to craft the HTTP-endpoint URLs for the following tests (see 6.9.4):

e) Apply all tests from 7.3.14 (see 6.9.3).

f) Create a suite of endpoint specific test cases according to 7.1.5. With those, verify that all endpoints behave as specified in this document. Use Table 52 as a guideline for some specific tests to perform, in addition to the general tests covering the endpoints, which are proposed in 7.1.5.

Table 52 — Proposed tests for endpoints of 4891-unit's direct messaging API server

Endpoint	Proposed tests
`GetUnitInformation` (see 6.9.3.2)	— Verify that the response contains the EUT unit's identifier, name, and unit certificate. — On the EUT, change the unit's name (if applicable), and verify that the new name is reflected in the response
`SendMessage` (see 6.9.3.3)	— Send 4891-messages with the destination unit being the EUT to this endpoint. Verify by observation, that the EUT signals the expected effects of the messages (as applicable to specified behaviour for the sent messages). — Send 4891-messages with the destination unit being a different 4891-unit to this endpoint. Verify that these messages are either forwarded further to another 4891 unit or to the 4891-message broker. This test is only applicable if EUT supports message relaying functionality. Consult the manufacturer's documentation to determine if that is the case.

7.3.4 Message broker client tests

7.3.4.1 General

The tests specified in 7.3.4 are applicable if the EUT connects to the 4891-message broker for publication or subscription of 4891-messages.

Verify that the EUT uses a MQTT-client compatible with protocol version `3.1.1` (see 6.5.3).

7.3.4.2 Connector configuration tests

For the tests a) and b) below, monitor the active client connections on the MQTT-server of the simulated 4891-message broker. Observe that the MQTT-client of the EUT connects to the server, showing up in the list of connected clients. Modify the configuration of the MQTT-server and update the relevant information in the message broker connector. Observe that the MQTT-client of the EUT reconnects to the MQTT-server, using the changed connectivity information eventually.

Perform the following tests:

a) Verify that the EUT uses the connector information for the 4891-message broker (i.e. MQTT host, port, username, password), that is propagated on the test environment, and configures its MQTT-client with that information (see 6.5.2 and 6.5.6).

b) Verify that the EUT adapts to the configuration changes of the connector (e.g. changes to the MQTT host, port, username, password), that are propagated on the test environment, and that the EUT reconfigures its MQTT-client with that information (see 6.5.2 and 6.5.6).

7.3.4.3 4891-message publication tests

This clause is applicable if the EUT publishes 4891-messages on the 4891-message broker.

For the tests specified in a) to e), follow the general test instructions of 7.1.7 to monitor the MQTT-messages published on the MQTT-server of the simulated 4891-message broker. Inspect the 4891-messages published on the 4891-message broker (e.g. connect a separate MQTT-client to the 4891-message broker and subscribe to the topic "`sappnet/#`") and only consider the messages published by EUT for the tests in this subclause.

Peform the following tests:

a) Verify that all 4891-messages contained in the published MQTT-messages are well-formed, i.e. apply all tests from 7.3.11 on those messages.

b) Verify that the MQTT-client of the EUT publishes MQTT-messages that contain well-formed 4891-messages, encoded correctly (see 6.5.4, and 6.5.5).

c) Verify that the MQTT-client of the EUT publishes the 4891-messages on their designated MQTT-topics, derived from the 4891-message headers (see 6.5.4, 6.5.5, and 6.5.5.2).

d) Verify that the MQTT-client of the EUT publishes the 4891-messages on topics that are lowercase (see 6.5.5.2).

e) Verify that the MQTT-client of the EUT encoded the published 4891-messages according to the JSON-encoding specified in 6.2.

7.3.4.4 4891-message subscription tests

This clause is applicable if the EUT subscribes to 4891-messages via 4891-message broker.

Consult the manufacturer's documentation to identify suitable message types for the tests in this subclause.

Craft suitable 4891-messages with content that the EUT is expected to be able to handle and react to. Encode those messages according to the JSON-encoding specified in 6.2. Use an MQTT-client to publish these messages on related MQTT-topics.

Perform the following tests:

a) Verify that the EUT subscribes to the MQTT-topics that are of interest for the EUT's type, and that those topics match the standardized topic patterns (see 6.5.6).

b) Verify that the EUT reacts in an expected way according to the manufacturer's documentation for the used message type and content.

7.3.5 Service discovery API client tests

7.3.5.1 General

The tests specified in 7.3.5 are applicable if the EUT interacts with endpoints of the 4891-service discovery API server.

Operate a simulated 4891-service discovery API server that will be contacted by the EUT for the tests specified in 7.3.5.2 to 7.3.5.3. On that service discovery, configure the connectors to the other components (e.g. message broker, unit registry, smart gateway unit) to simulated versions of those, so that the EUT can also contact those via their interfaces if applicable.

7.3.5.2 API base URL configuration tests

Perform the following tests:

a) Verify that the manufacturer's documentation states how the connection settings to the 4891-service discovery API can be configured, or if that configuration is picked up from UDP discovery packets (see 6.6.3).

b) If configuration happens automatically, apply all tests from 7.3.13 with the parametrization specified in 6.6.5.

7.3.5.3 API server communication tests

Configure the API client of the EUT according to manufacturer's documentation, so that the EUT uses the simulated API server for the tests in this subclause.

Perform the following tests:

a) Apply all tests from 7.3.15 (see 6.6.3).

b) Verify by inspection, that the EUT requests the service connectors from the API server (see 6.6.3). The EUT is expected to initiate communication with some or all the other services configured in the service discovery. Verify that the EUT has picked up the service connectors from the API response and has started contacting the other service components with the configured connectivity settings.

c) Modify the configured service connectors in the simulated service discovery.

Wait for the EUT to request the connectors again or trigger any functionality on EUT that can invoke this reloading. Consult the manufacturer's documentation to find which functionality results in connectors being reloaded. Verify that the EUT has picked up those new service connectors, contacting the other services with the modified connectivity settings.

7.3.6 Unit registry API client tests

7.3.6.1 General

The tests in 7.3.6 are applicable if the EUT interacts with endpoints of the 4891-unit registry API server.

Operate a set of simulated 4891-components (e.g. their API servers), that can be contacted by the EUT as part of the tests of 7.3.6. A simulated 4891-unit registry API server is especially useful for those tests.

7.3.6.2 API server communication tests

Configure the connectivity settings to the 4891-unit registry API server on the EUT. Consult the manufacturer's documentation to determine if that is accomplished automatically (i.e. the EUT picking up the connector from the 4891-service discovery) or if it is required to be configured manually.

Perform the following tests:

a) Apply all tests from 7.3.15 (see 6.7.3).

b) Verify that the EUT is fetching the active root certificates from the simulated 4891-unit registry API server eventually (see 6.7.3).

c) If the EUT represents a 4891-unit, verify that the EUT is requesting on API endpoints related to the unit registration process on the simulated API-server (see 6.7.3).

d) Repeat tests a) to c), but this time respond with malformed HTTP-responses from the simulated API server. Verify that the EUT is handling those failure situations, e.g. eventually retrying requests and not ending in unrecoverable error states.

7.3.7 Direct messaging API client tests

7.3.7.1 General

The tests in 7.3.7 are applicable if the EUT can connect to another 4891-unit's direct messaging API server, e.g. looking up those unit's meta information or sending 4891-messages directly to it.

Operate a simulated 4891-unit with a running direct messaging API server that will be contacted by the EUT for the tests in 7.3.7. As the EUT can require other 4891-components to be available in the 4891-network, make sure to run simulated instances of those other components as well, where applicable, e.g. a 4891-unit registry or 4891-service discovery.

Verify that the manufacturer's documentation states how the direct messaging functionality of the EUT can be activated and how it is configured.

7.3.7.2 API base URL configuration tests

Perform the following test:

a) If the direct messaging connectivity settings are picked up from UDP packets, then apply all tests from 7.3.13 with parametrization as specified in 6.9.4, pointing to the simulated unit as direct messaging API server.

7.3.7.3 API server communication tests

Configure the API client of the EUT according to manufacturer's documentation, so that the EUT uses the simulated API server (see 6.9.2).

Perform the following tests:

a) Apply all tests from 7.3.15 (see 6.9.2).

b) Verify by inspection, that the EUT invokes the API endpoints on the direct messaging API server of the simulated unit. Consult the manufacturer's documentation to determine how the EUT can be provoked to perform these interactions, or if EUT is triggering those automatically. The following behaviour can be expected from the EUT:

— the EUT is requesting information about the receiving unit via endpoint `GetUnitInformation` (optional);

— the EUT is sending a message to the endpoint `SendMessage` (once per relayed message).

c) Verify by inspection that the provided requests conform to the API specification.

7.3.8 Message relaying tests

The tests in this subclause are applicable if the EUT supports message relaying functionality. Consult the manufacturer's documentation to determine if that is the case for the EUT and how that functionality is activated.

Consult the manufacturer's documentation to determine if relayed messages are stored on EUT and delivered with a delay in any way. Identify specified behaviour on how these stored messages are delivered or how long to wait if the delay cannot be circumvented. The tests in this subclause assume immediate delivery of a relayed message. If there is any delay, wait for the delivery to happen according to documented behaviour of EUT.

Create any well-formed 4891-message with a destination other than the EUT, i.e. set "`DestUnitId`" to a unit identifier not used to identify the EUT itself. Send that message to the EUT in any applicable way. Perform multiple tests according to Table 53, each time varying the created message with regards to the "`ExpiresAt`" message header (see 5.4).

Consult the manufacturer's documentation, to determine which ways of message relaying are supported by EUT. For test cases where the message is expected to be delivered, verify that at least one of the following ways of delivery is performed by EUT (see 6.9):

a) The message is published on the simulated 4891-message broker by EUT, addressed to the destination unit.

b) The message is delivered on the direct messaging API of a simulated 4891-unit acting as destination unit and the delivery is performed from the EUT.

c) The message is delivered on the direct messaging API of a simulated 4891-unit acting as an unrelated unit (see 6.9). If this case passes, verify that the manufacturer's documentation states how the circular delivery of messages is mitigated.

Table 53 — Tests for discarding behaviour of message relaying functionality of 4891-component

Test no.	`ExpiresAt` header	Expected behaviour
1	do not set header	the message is eventually delivered
2	a value in the future (after potential delay of caching message on EUT before delivery)	the message is eventually delivered
3	a value in the past	the message is not delivered

7.3.9 Root certificate properties tests

Inspect the root certificate for the tests in this subclause.

Perform the following tests:

a) Verify that the certificate is structured in X.509 format (see 6.10.2).

b) Verify that the key-pair associated with the certificate is based on one of the following cipher algorithms (see 6.10.2):

- RSA
- DSA
- ECDSA

c) Verify that the signature of the certificate uses one of the following hashing algorithms (see 66.10.2):

- SHA-256
- SHA-384

d) Verify that the "`Common Name`" property matches the string "`sappnet:ureg`" (see 6.10.3.1).

7.3.10 Unit certificate properties tests

Inspect the unit certificate for the tests in this clause.

Perform the following tests:

a) Verify that the certificate is structured in X.509 format (see 6.10.2).

b) Verify that the key-pair associated with the certificate is based on one of the following cipher algorithms (see 6.10.2):

- RSA
- DSA

— ECDSA

c) Verify that the signature of the certificate uses one of the following hashing algorithms (see 6.10.2):

— SHA-256

— SHA-384

d) Verify that the "`Common Name`" property is set to "`sappnet:unit:<uuid>`", with "`<uuid>`" being a string-encoded UUID in "8-4-4-4-12" format, representing a unit identifier (see 6.10.4.1). For example:

— `sappnet:unit:ffd3b208-d488-481c-9476-7ea7552f0a05` (full value)

— `ffd3b208-d488-481c-9476-7ea7552f0a05` (the `<uuid>` part)

e) Verify that the "`<uuid>`" in the "`Common Name`" property matches the unit identifier of the related unit (see 6.10.4.1).

f) Verify that the "`Subject`" property contains a public-key that matches the public-key of the related unit (see 6.10.4.1).

g) Verify that the "`Issuer`" property references the main root certificate that is configured on the 4891-unit registry at the time of issuing (see 6.10.4.1).

h) Verify that the certificate is signed by the private-key of the referenced root certificate, i.e. the signature can be validated with the public-key of the referenced root certificate (see 6.10.4.1).

7.3.11 4891-message properties tests

The tests of this subclause are applicable to all 4891-messages created by EUT. To inspect the created messages in the tests a) to i), use a simulated direct messaging API server, a simulated 4891-message broker or inspect the network traffic.

Apply the tests in this clause on each 4891-message that is received from the EUT as part of any other test.

Perform the following tests:

a) Verify that the message is well-structured, i.e. it follows the structural format composition as specified in the document (see 5.3.2).

b) Verify that the message's header names are well-formed as specified in this document (see 5.3.3).

c) Verify that the message does not contain reserved standard header names, i.e. it does not have any header not starting with an "`X`" or "`x`" character that is not specified anywhere in this document or other standards (see 5.3.3).

d) If the message contains any non-standard headers (starting with an "`X`" or "`x`" character), then verify that those also contain a manufacturer-specific prefix as part of their name after that character, e.g. "`XManufacturerSomething`" (see 5.3.3).

e) Verify that the message has the following headers with values valid for the respective headers (see 5.3.3.2):

— `Id`

— `Type`

— `SrcUnitId`

— `CreatedAt`

f) If the message is a response message created by EUT as a reaction to a previously received request message, then verify that it has a `ReferenceId` header. Verify that the value of that header matches the value of the `Id` header of the request message (see 5.3.3.2).

g) If the message contains parts, then verify that each of those has these properties:

   — The part has an `Encoding` header with valid value (see 5.3.3.3).

   — The part's structure matches that declared encoding (see 5.3.4).

h) If the encoding is declared as `RECORD`, then verify that the part's data are following the record definition as specified in the documentation of that message type (see 5.3.4.2). If the message type is standard, look up the definition in the related standard. If the message type is non-standard, look up the definition in the manufacturer's documentation.

i) Verify that the value of the `Type` header has exactly one of the following properties (see 5.3.5):

   — either starts with an "`X.`" or "`x.`" prefix, denoting it to be a non-standard message type. In this case, the type also contains a manufacturer-specific prefix, e.g. "`x.manufacturer.something`".

   — or it is a standard type, i.e. specified in this document or another standard.

7.3.12 UDP discovery broadcast sending tests

This clause is applicable if the EUT broadcasts UDP discovery packets containing information about a functionality exposed to the network by the EUT (e.g. connectivity information about an HTTP-API server).

Perform the following tests:

a) Verify by inspection that the UDP packet is sent to UDP port 4891 (see 6.4.1).

b) Verify by inspection that the UDP packet's content matches the specified format grammar (see 6.4.1).

c) Verify by inspection that the UDP packet's "`packet-type`" token contains only valid characters (see 6.4.1).

d) Verify by inspection that the UDP packet's content is encoded in UTF-8 (see 6.4.1).

7.3.13 UDP discovery broadcast listening tests

This subclause is applicable if the EUT listens to UDP discovery packets to automatically configure connectivity information to network-exposed services of the packet-emitting equipment.

To perform the tests in this subclause, identify the specific UDP discovery packets that EUT listens to. For each kind of discovery packet identified that way, apply the following tests to EUT:

a) Verify that the EUT reacts to applicable discovery packets broadcast to UDP port 4891 (see 6.4.2). For example, all 4891-components should listen to "4891-service discovery API"-discovery UDP packets and update their internal configuration. Craft a suitable UDP packet and broadcast it into the network to which the EUT is connected. Verify by observation that the EUT reacts to the packet's content in an expected way (e.g. trying to connect to simulated HTTP-API server endpoints for API-discovery packets).

b) Perform distinct test runs for the tests listed in Table 54. For each test, operate a simulated HTTP-API server, the kind that the EUT is expected to interact with (e.g. "service discovery API"), and broadcast a manually crafted UDP discovery packet for the simulated HTTP-API server (e.g. a "service discovery API" discovery packet) in the network shared by the simulated server and the EUT.

Table 54 — Tests for handling of UDP discovery packets

Test no.	Properties of the crafted UDP packet	Expected EUT behaviour
1	set "`packet-header`" all upper-case characters, e.g. "`SAPPNET:SD`"	EUT configures correctly and eventually contacts the API server to trigger functionality
2	set "`packet-header`" all lower-case characters, e.g. "`sappnet:sd`"	EUT configures correctly and eventually contacts the API server to trigger functionality
3	set "`packet-header`" with mixed-case characters, e.g. "`sappNet:Sd`"	EUT configures correctly and eventually contacts the API server to trigger functionality
4	add additional packet params that are not specified for the used packet-type	EUT either ignores the packet entirely or it behaves as if the extra parameters are not provided
5	set "`packet-type`" to an unexpected value, e.g. "`foo`"	EUT ignores the packet because it is an unexpected packet-type
6	start the packet with the expected "`packet-header`", but add random data after it	EUT ignores the packet because it is malformed

7.3.14 HTTP-API server tests

7.3.14.1 General

The tests of 7.3.14 are applicable for all HTTP-API servers operated by the EUT.

Perform the following tests for each HTTP-endpoint exposed by the HTTP-API server operated by the EUT:

a) Verify that the HTTP-API server endpoints expect incoming data (HTTP-request body, HTTP-request query parameters) to be encoded according to the JSON-encoding scheme specified in 6.2. Verify that malformed and differently encoded content is producing error responses on all documented endpoints.

b) Verify by inspection that the HTTP-API server endpoints produce outgoing data (HTTP-response body) that is encoded according to the JSON-encoding scheme specified in 6.2.

c) Verify by inspection that the HTTP-API server supports HTTP-protocol version `1.1` (see 6.3).

d) Verify by inspection that produced HTTP-responses with content body have HTTP-Header "`Content-Type`" set to "`application/json`" (see 6.3.4). Verify that the content body is encoded in JSON.

e) Verify by inspection that produced HTTP-responses indicating success and not having a content body have an HTTP-status of 204 (see 6.3.4).

f) Verify by inspection that produced HTTP-responses indicating success and having a content-body have an HTTP-status of 200 (see 6.3.4). Also verify that such responses have HTTP-Header "`Content-Type`" set to "`application/json`".

g) Verify by inspection that produced HTTP-responses indicating errors have an HTTP-status of 400 to 999 (see 6.3.4). Also verify that such responses have HTTP-Header "`Content-Type`" set to "`application/json`" and that their content body contain an error description in expected format (see 6.3.5).

h) Produce an HTTP-request that should trigger an error situation on the specific HTTP-API endpoint under test. Send that request and verify that the HTTP-API-server produces an error response (see 6.3.5).

7.3.14.2 4891-unit authentication-protected endpoint tests

The tests in this clause are applicable to all HTTP-API endpoints, that are protected with unit authentication (see 6.3.6).

For the endpoint under test, verify that all tests in this clause succeed (see 6.3.6.4).

Perform the following tests:

a) Verify that the endpoint is not able to be used without the authentication handshake. To do so, create a valid HTTP-request as specified on the endpoint's documentation, but instead of modulating it with the authentication handshake, send it to the endpoint directly, as if the endpoint is “non-protected”. Expect the server to react with an error response.

b) For the tests in Table 55, prepare a valid HTTP-request again, but this time modulate it through the authentication handshake as specified for protected HTTP-API endpoints. Each of those tests replaces certain parts of that modulation process to test the server's behaviour. Verify for each of those tests, that the server reacts in the expected way.

Table 55 — Additional tests for 4891-unit authentication-protected endpoint

Test no.	Modifications on the request	Expected result
1	do not set the “`unitId`” field	error “`INVALID_REQUEST`”, because of missing unit identifier
2	do not set the “`signature`” field	error “`INVALID_REQUEST`”, because of missing signature
3	set the “`unitId`” field to a random id value (non-registered unit)	error “`FORBIDDEN`”, because of unit certificate mismatch
4	set the “`unitId`” field to the identifier of another registered unit, that is not the one that produced the signature, i.e. not matching the unit certificate used for the signature	error “`FORBIDDEN`”, because of unit certificate mismatch
5	change the value of “`requestContent`” after computing the signature	error “`FORBIDDEN`”, because of signature mismatch
6	allow the used auth-token to expire by waiting more than 60 s before sending the request using that token	error “`FORBIDDEN`”, because of expired auth-token
7	set the “`authToken`” field to an auth-token that has already been used by a former successful request.	Error “`FORBIDDEN`”, because of invalid/reused auth-token

7.3.15 HTTP-API client tests

7.3.15.1 General

The tests in 7.3.15 are applicable if the EUT operates an HTTP-API client interacting with an HTTP-API server endpoint as specified in this document.

For testing, operate a simulated HTTP-API server of a kind that the EUT will interact with.

Perform the following tests by inspecting the requests sent from the EUT to the simulated API server:

a) Verify that the request uses HTTP protocol version `1.1` (see 6.3).

b) Verify that the request's HTTP-header “`Accept`” is set to “`application/json`” (see 6.3.2).

c) If the request contains a content body, verify that the request's HTTP-header “`Content-Type`” is set to “`application/json`” (see 6.3.2).

d) If the request contains a content body, verify that the content body is a valid JSON-encoded string (see 6.3.2).

e) If the request uses query-parameters in the URL/path, verify that those are properly encoded (see 6.3.3).

Perform the following tests to verify that the EUT handles error responses (see 6.3.5):

f) If applicable, send an error-response back to the EUT for an incoming request on the simulated API-server. Use an error-code as described in the specification for the API endpoint. Verify that the EUT behaves as expected for that error-situation.

g) Repeat the test specified in f), but this time use an error-code for the response that is not specified in the API endpoint's specification. Verify that the EUT behaves in an appropriate way, i.e. not crashing or locking in an unrecoverable state.

7.3.15.2 4891-unit authentication-protected endpoint tests

The tests in this clause are applicable if the EUT operates an HTTP-API client interacting with an HTTP-API server endpoint that is protected via unit authentication as specified in this document (see 6.3.6).

For testing, operate a simulated HTTP-API server of a kind, with which the EUT will interact.

Perform the following tests by inspecting the requests sent from the EUT to the simulated API server (see 6.3.6.3):

a) Verify that the EUT initiates the authentication handshake by requesting an auth-token from the API, before the protected endpoint is contacted. Note down a new random value for the auth-token and send it as the response to the request. Make sure to not respond with the same value twice.

b) Verify that the EUT sends a request to the protected endpoint, with the request's content being a valid JSON-encoded string.

c) Verify that the request content contains a "`signature`" field, containing a Base64-encoded string and decode it (signature).

d) Verify that the request content which contains a "`data`" field is a valid JSON string and decode it (data-structure).

e) Verify that the "`unitId`" field in the data-structure contains the identifier of the EUT (unit identifier).

f) Verify that the "`authToken`" field in the data-structure contains the noted down value from first step.

g) Verify that the "`requestContent`" field contains a valid JSON-encoded string and decode it (content-structure).

h) Verify that the content-structure matches the specification of the invoked API endpoint.

i) Lookup the unit certificate for the received unit identifier on the simulated 4891-unit registry on which the EUT registered. Verify that the received signature is valid for the data-string of the request against the looked-up unit certificate.

Annex A
(normative)

Smart gateway — Interface to controlled equipment

A.1 General

This annex contains requirements for the implementation of 4891-smart gateway units which are intended to integrate with IEC 61162-450/ IEC 61162-460-equipment.

The IEC 61162-450/ IEC 61162-460-equipment is called “controlled equipment” in this document.

Two HTTP-APIs are specified as interface between the 4891-smart gateway unit and the controlled equipment (see Figure A.1). Those APIs provide functionality to exchange messages based on 4891-messages. This annex also specifies a way to authenticate a user securely against the controlled equipment, if applicable for the controlled equipment.

The specific message types to be exchanged via this interface and the requirement for user authentication depend on specific use cases and type of controlled equipment to be integrated. Those message types and the need for authentication are specified in use-case-specific requirements (e.g. “smart logbook” in Annex A) or manufacturers’ documentations.

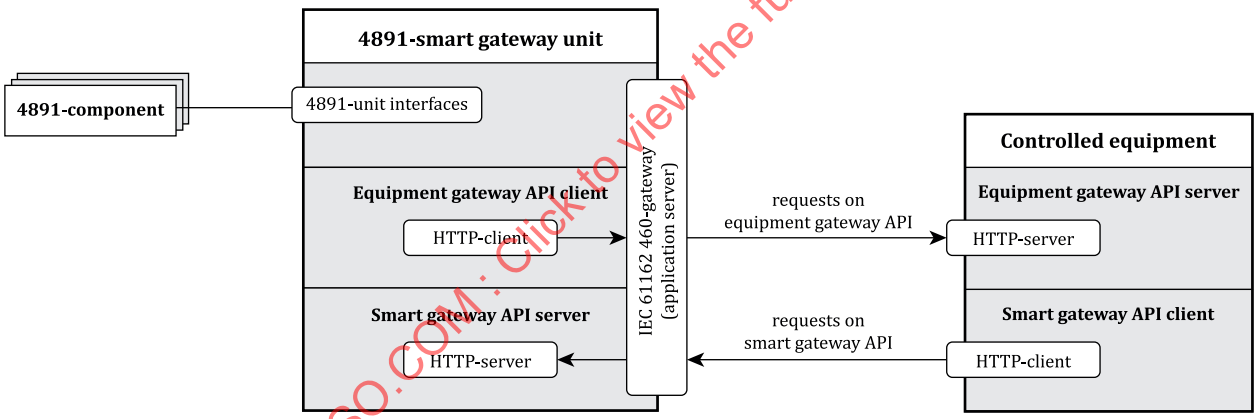


Figure A.1 — Interface between 4891-smart gateway unit and controlled equipment

A.2 Interface between 4891-smart gateway unit and controlled equipment

A.2.1 General

To avoid additional traffic to the transmission groups defined in the navigation network described in IEC 61162-450, all non-navigational traffic shall be separated from it.

This may be implemented in any applicable way, for example:

— VLAN according to IEC 61162-460;

— own IP subnet.

For any reading and writing of messages between the 4891-smart gateway unit and the connected 450/460 equipment, “Other Network Functions” (ONF) shall be used. These messages shall be encoded in JSON as specified in 6.2.4.

A.2.2 IEC 61162-460 gateway

The 4891-smart gateway unit shall either provide a gateway in accordance with IEC 61162-460, or it shall be connectable to a gateway in accordance with IEC 61162-460.

That gateway will be used to allow communication between 4891-components and controlled equipment. The gateway provides a DMZ covert by two firewalls. Inside the DMZ, an application server handles the data transfer through the gateway (see Figure A.1).

A.2.3 Application server

The application server acts like a converter. Each received request shall be decoded, analysed, and re-encoded again by the application server, before being sent to the destination equipment. The application server shall not directly forward received requests without passing them through those steps. In the analysing-step, the application server shall validate the request's structure and content, by the following rules:

— Requests shall have expected headers and well-formed content body, as defined in the API specifications of this document;

— For requests containing 4891-messages:

  — the message type shall be one of the expected types,

  — the message headers and parts are of expected structure and content;

— Values have expected types and data format (depending on message definitions);

— Values are in semantically sound data ranges (e.g. most timestamps should not be in the future, amounts are not negative, …).

A.2.4 External firewall

If a separate gateway is used, the external firewall shall be configured with the 4891-smart gateway unit's MAC address.

If the gateway is part of the 4891-smart gateway unit, a separate network card shall be used to connect to the network in accordance with IEC 61162-460. No other equipment shall be connected to that network card.

A.3 Data security

The communication between the 4891-smart gateway unit and the controlled equipment shall be non-encrypted, so that the application server and gateway are able to inspect and translate the data (see A.2).

The communication between the 4891-smart gateway unit and 4891-I/O units with regards to the messages originating from or destined to the controlled equipment shall be encrypted. For this, the 4891-message encryption shall be used:

a) Messages originating from controlled equipment shall be encrypted for the destined 4891-I/O unit by the 4891-smart gateway unit before sending it to that unit directly or indirectly.

b) Messages originating from 4891-I/O units with the intent to forward them to the controlled equipment (likely containing user credentials) shall be encrypted by the sending I/O unit for the 4891-smart gateway unit as a destination.

Refer to 6.10 for details on data trust, encryption and digital signatures.

A.4 4891-smart gateway unit as gateway to controlled equipment

A.4.1 General

4891-I/O units shall not exchange 4891-messages with controlled equipment directly. Instead, the 4891-smart gateway unit shall act as a messaging gateway between the I/O units and the controlled equipment.

A.4.2 Gateway identifiers

Controlled equipment, that is connected to the 4891-smart gateway unit and shall be addressable by 4891-I/O units, shall have a unique gateway identifier associated to it.

These gateway identifiers shall be used as values for the `SrcGateway` and `DestGateway` message headers (see A.9) to identify the controlled equipment that the 4891-I/O units communicate with (see A.4.5 and A.4.6).

Gateway identifiers shall be composed only of these characters:

— `a` to `z` (lowercase letters)

— `A` to `Z` (uppercase letters)

— `0` to `9` (numeric digits)

— `.` (point)

Gateway identifiers shall be handled case-insensitive, e.g. identifiers such as `X.SOME.Equipment` and `x.some.equipment` shall be considered synonyms for the same gateway.

Like message types, the gateway identifier's value space is divided into two groups:

a) An identifier not beginning with the `x` character is a "standard gateway identifier".

b) An identifier beginning with the `x` character is a "non-standard gateway identifier".

The gateway identifier of a specific controlled equipment shall be documented together with the set of supported message types that can be exchanged with that equipment.

This annex does not define specific gateway identifiers, instead those can be found in:

— Annex B, which specifies a gateway identifier for integration with the ELRB of ISO 21745;

— other standards defining standardized extensions for this document, i.e. specifying integrations with standardized equipment;

— manufacturer's documentation for non-standard extensions, i.e. specifying integrations with proprietary equipment.

A.4.3 Configuration of available gateways

The list of available gateways for connected equipment shall be configurable in the 4891-smart gateway unit. Implementations may have a manual configuration for this or collect available gateways automatically when connecting to supported controlled equipment.

The manufacturer's documentation shall describe how the gateway configuration can be performed.

A.4.4 Listing available gateways

The 4891-smart gateway unit shall implement functionality to list the available configured gateways.

For this, 4891-I/O units shall send a message of type `smgw.gateways.query` to the smart gateway unit, expecting a response message of type `smgw.gateways.queryresp` (see A.10.2).

A.4.5 Sending message to controlled equipment from 4891-I/O unit

To send a message to controlled equipment, the 4891-I/O unit shall put the following values on the message headers:

— The `DestUnitId` message header shall be set to the 4891-smart gateway unit's unit identifier.

— the `DestGateway` message header shall be set to the designated gateway identifier for the equipment that should receive the message.

The value of the `DestGateway` header identifies which controlled equipment the message is intended to be communicated to by the 4891-smart gateway unit on behalf of the sending 4891-I/O unit.

A.4.6 Receiving message from controlled equipment on 4891-I/O unit

4891-I/O units can detect that a 4891-message originates from controlled equipment by inspecting that message. The following conditions shall be met for a message to be considered as "sent by controlled equipment":

— the `SrcUnitId` message header has the unit identifier of the 4891-smart gateway unit;

— the `SrcGateway` message header exists.

The value of the `SrcGateway` header identifies the controlled equipment by which the message has been created.

A.4.7 Error logging

The 4891-smart gateway unit shall provide functionality for logging error messages. The error logging functionality shall follow IEC 61162-450:2018, 4.3.3.

The error messages listed in Table A.1 shall be supported, if syslog (see RFC 5424) is adapted as described in IEC 61162-450:2018, 4.3.3.2. Table A.1 also specifies the MSGID (see RFC 5424) to be used.

Table A.1 — Types of errors to be logged via syslog

MSGID	Error	Data to collect
`sappn.eg.reach`	controlled equipment not reachable via equipment gateway API	— destination gateway identifier
`sappn.eg.authmiss`	missing username/password when authentication is required via equipment gateway API	— destination gateway identifier — readable description of the error — message type (if applicable)
`sappn.eg.authinv`	invalid username/password when authenticating via equipment gateway API	— destination gateway identifier — readable description of the error — message type (if applicable)
`sappn.eq.error`	failing requests on the equipment gateway API	— destination gateway identifier — readable description of the error
`sappn.cg.error`	failing requests on the smart gateway API	— source gateway identifier — readable description of the error

A.5 Authorized operations

A.5.1 General

Certain operations on controlled equipment can require an authorized user performing the operation. Such operations are called “authorized operations” (AOP).

It shall be specified whether an operation shall be considered an AOP or not, together with the message types. If not stated explicitly, a message type shall be considered as not part of an AOP.

A.5.2 AOP messaging

4891-messages which are part of an AOP and which are sent to controlled equipment from a 4891-I/O unit, shall have two additional headers containing the credentials of a user authorized for that AOP on the controlled equipment: `AuthUsername`, `AuthPassword`.

The 4891-smart gateway unit shall inspect the `AuthUsername` and `AuthPassword` header fields in incoming 4891-messages. In case they are set, the smart gateway unit shall use them to perform the authentication challenge computation as described in A.6 and A.7.

The 4891-smart gateway unit shall provide the computed authentication challenge result together with the message to the controlled equipment for processing of the AOP message.

The controlled equipment shall validate the received message and authentication proof before performing any logic.

See Figures A.7 to A.15 for communication examples for different authentication and authorization situations in the context of AOP messages.

A.5.3 AOP mode

A.5.3.1 General

4891-I/O units shall differentiate between the two modes of operation:

a) regular operation mode;

b) authorized operation mode (AOP mode).

By default, 4891-I/O units shall operate in regular operation mode. The AOP mode is considered active when the I/O-unit has knowledge of user credentials that allows authentication against a controlled equipment.

A.5.3.2 Entering AOP mode

4891-I/O units intending to enter AOP mode shall query user authentication data from the operating user (i.e. username and password). Only when providing valid user credentials, the I/O unit shall be able to switch to AOP mode.

To validate the provided user credentials, the I/O unit shall send them in a message of type `smgw.users.auth` to the 4891-smart gateway unit with `DestGateway` message header containing the gateway identifier of the designated controlled equipment (see A.4.2).

It is permitted for 4891-I/O units to locally persist the user credentials. In that case, those credentials shall be encrypted with a user defined PIN of five or more digits. When the user enters the correct PIN, the I/O unit shall behave as if the user has entered the stored credentials. When the user enters an incorrect PIN five times in a row, the I/O unit shall delete the stored credentials, falling back to the regular user credentials input for entering the AOP mode.

A.5.3.3 Leaving AOP mode

4891-I/O units shall automatically leave AOP mode after the user has ended the interaction with the unit. The following list describes indicators of inactivity which shall be applied, where applicable:

— display or screen of the unit has been locked (i.e. lock screen or screensaver);

— user has been signed out (explicitly or automatically);

— 1 min has passed without user inputs (e.g. keyboard, mouse movement, touch events).

The semantics of “end of interaction” or “inactivity” is not strictly defined. Manufacturers shall use the indicators for inactivity listed above to derive suitable definitions for their implementation.

A.5.4 Handling of user credentials

The 4891-smart gateway unit may receive user credentials in the `AuthUsername` and `AuthPassword` headers of incoming messages from 4891-I/O units.

The 4891-smart gateway unit shall not persist those user credentials any longer than needed to perform the authentication processes against the controlled equipment (see A.7).

The 4891-I/O units may receive user credentials as input from the authorized users directly to forward them to the 4891-smart gateway unit for authentication against controlled equipment (e.g. as part of AOP mode).

4891-I/O units shall not persist those user credentials for any reason other than for enabling the AOP mode and transmitting the 4891-I/O units to the 4891-smart gateway unit for the authentication processes against the controlled equipment. Moreover, 4891-I/O units shall not share those user credentials with any other system. It is permissible to wrap the 4891-message containing the user credentials in an encrypted message (type `base.encrypted`) for sending the message to the 4891-smart gateway unit through the 4891-message broker or other 4891-I/O units (message relaying). See A.3 for further details.

A.6 Extended messaging processes

A.6.1 General

A.6.2 to A.6.5 specify extensions to the general messaging processes specified in 6.11. Manufacturers shall implement the extensions specified on 4891-components.

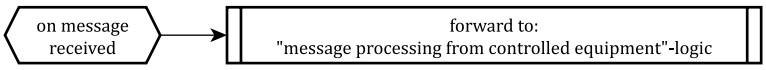
A.6.2 Receiving 4891-messages from controlled equipment

The 4891-smart gateway unit can receive 4891-messages in two ways from the controlled equipment:

a) the controlled equipment responds with a message synchronously to a request on its equipment gateway API (see Figure A.2);

b) the controlled equipment sends a message asynchronously via smart gateway API (see Figure A.3).

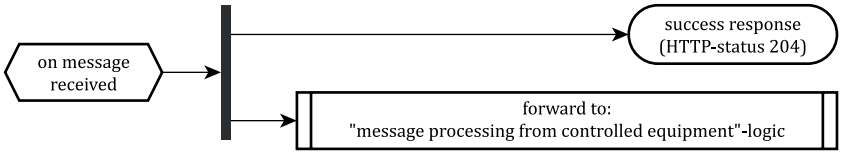
In both cases, the received message shall be processed by the “message processing from controlled equipment”-logic (see Figure A.4) before being forwarded into the “general message processing”-logic (see 6.11.5).



NOTE 1 This logic is executed when receiving a message from controlled equipment in response to a request on its equipment gateway API.

NOTE 2 See Figure A.4 for referenced process “message processing from controlled equipment”-logic.

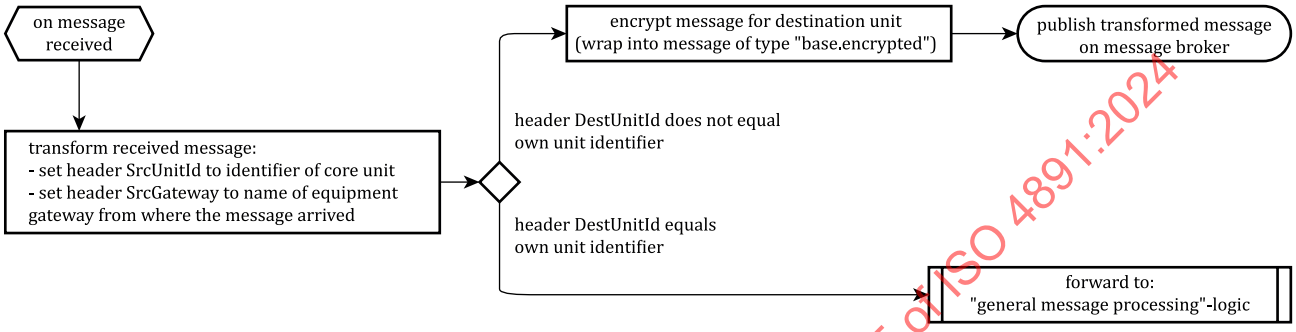
Figure A.2 — Smart gateway unit — Message received in response of equipment gateway API



NOTE 1 This logic is executed when receiving a message from controlled equipment on the smart gateway API.

NOTE 2 See Figure A.4 for referenced process “message processing from controlled equipment”-logic.

Figure A.3 — Smart gateway unit — Message received on smart gateway API



NOTE 1 This logic is executed for messages received from the controlled equipment on the 4891-smart gateway.

NOTE 2 See 6.11.5 for referenced process “general message processing”-logic.

Figure A.4 — Smart gateway unit — Message processing from controlled equipment

A.6.3 Message handling logic for smart gateway unit

The smart gateway unit shall implement a message handling logic based on Figure A.5. This logic provides a process for the basic handling of messages received from I/O units that are intended to be delivered to the controlled equipment.

This logic delegates the message into the logic to forward it to the controlled equipment (see A.6.4) if the destination gateway has been specified on the message. In any case, the message is also forwarded into an implementation specific handling logic.

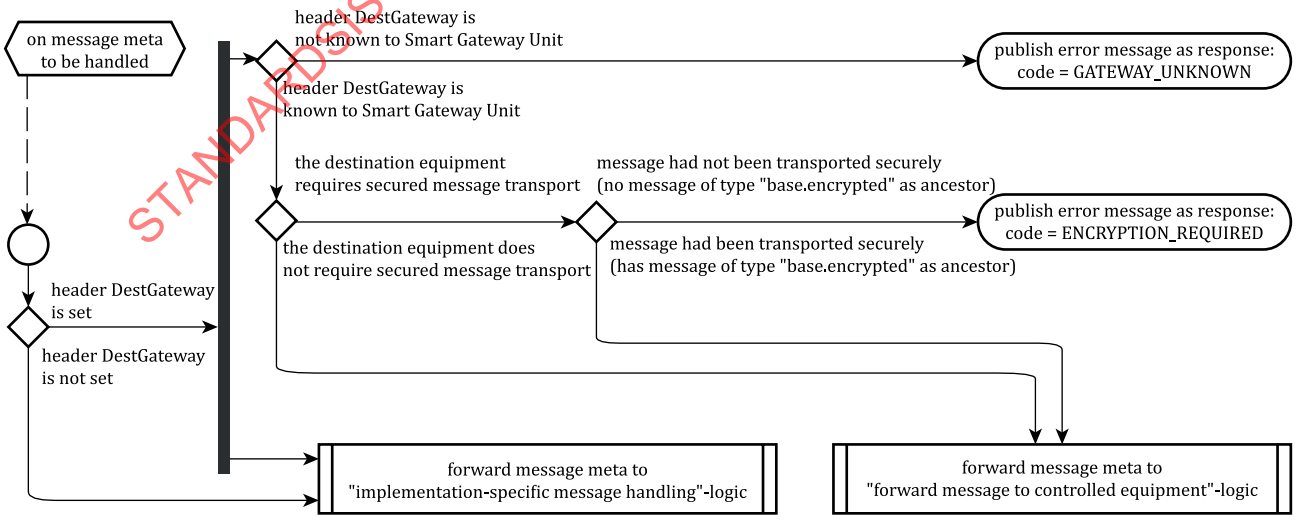


Figure A.5 — Smart gateway unit — Message handling

The smart gateway unit shall implement the logic from Figure A.6 for forwarding messages to the controlled equipment via equipment gateway API (see A.7) exposed by that equipment. The smart gateway unit shall only forward messages based on its message handling logic, as described in previous paragraph.

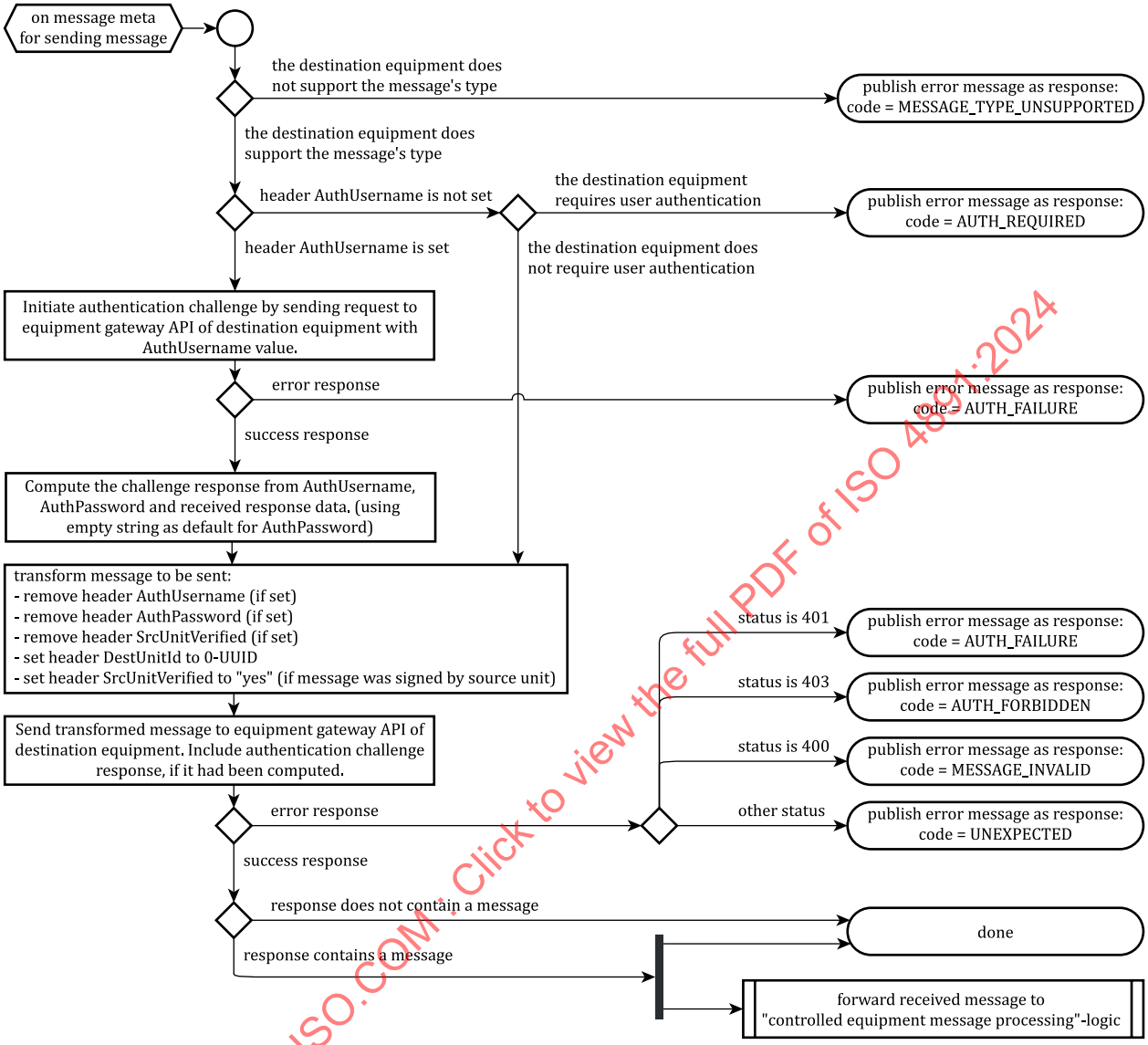


Figure A.6 — Smart gateway unit — Forward message to controlled equipment

A.6.4 Message properties and transformation

A.6.4.1 General

A.6.4.2 to A.6.4.5 specify required properties of 4891-messages in the context of message exchange with controlled equipment.

A.6.4.2 Messages from 4891-I/O units to 4891-smart gateway unit

All 4891-messages intended for delivery to the controlled equipment shall comply with these properties, before being sent form the creating 4891-I/O unit:

a) Message header fields have these values:

- `SrcUnitId`: identifier of the 4891-unit performing the operation.
- `DestUnitId`: set to identifier of smart gateway unit.
- `DestGateway`: set to the gateway identifier of the designated controlled equipment.

b) Message header fields (if the operation requires user authentication):

- `AuthUsername`: set to username credential of operating user, the operation will be performed in the name of that user.
- `AuthPassword`: set to password of operating user.

c) Message may be wrapped in a message of type `base.signed`, being signed by the creating 4891-I/O unit. This is required for the message to be flagged as "`SrcUnitVerified`" when delivered to controlled equipment. Manufacturer's documentation shall state if this flag is required for specific functionality of controlled equipment.

d) Message is wrapped in a message of type `base.encrypted`, being encrypted for the 4891-smart gateway unit.

A.6.4.3 Transformation of messages from 4781-smart gateway unit to controlled equipment

All 4891-messages sent from 4781-smart gateway unit to the controlled equipment shall be transformed by the smart gateway unit in the following way before they are sent:

a) remove message headers:

- `AuthUsername` (if set)
- `AuthPassword` (if set)
- `SrcUnitVerified` (if set)

b) set message header field:

- `DestUnitId = 00000000-0000-0000-0000-000000000000` (0-UUID)

c) set message header field, if the message is a verified message, i.e. if it was sent to 4891-smart gateway unit wrapped in a message of type `base.signed` with a valid signature signed by the same 4891-unit that is stated in the `SrcUnitId` header:

- `SrcUnitVerified = yes`

A.6.4.4 Messages from controlled equipment to 4891-smart gateway unit

All 4891-messages created by the controlled equipment shall have the following properties before being sent to the 4891-smart gateway unit:

a) set message header fields:

- `SrcUnitId = 00000000-0000-0000-0000-000000000000` (0-UUID)
- `SrcGateway`: the value from the header `DestGateway` of the corresponding request message that this new message is the response to, i.e. the gateway identifier assigned to the controlled equipment.
- `DestUnitId`: the value from the header `SrcUnitId` of the corresponding request message that this new message is the response to.

— `ReferenceId`: the value from the header `Id` of the corresponding request message that this new message is the response to (only if the message is a response).

A.6.4.5 Transformation of messages from controlled equipment on 4891-smart gateway unit

The 4891-smart gateway unit shall transform all 4891-messages received from the controlled equipment in the following way before they are sent to the intended target I/O unit:

a) set message header fields:

— `SrcUnitId`: the unit identifier of the smart gateway unit

b) wrap the message into a message of type `base.encrypted` encrypted for the destination I/O-unit.

A.6.5 Exchanging messages between 4891-I/O unit and controlled equipment

The following steps summarize the process that is performed when exchanging messages between 4891-I/O units and controlled equipment through the 4891-smart gateway unit. These steps are extracted from the process diagrams specified in Figures A.2 to A.6. Manufacturers shall implement the process in accordance with Figures A.2 to A.6, rather than following the simplified summary of steps.

For messages in operations that do not require an authorized user (regular mode of I/O unit), the SCRAM related steps and authentication related values are skipped. The following list assumes that the messages being exchanged are part of an authorized operation (AOP mode of I/O unit).

a) I/O unit enters AOP mode.

b) I/O unit composes a message that is supported by designated controlled equipment.

c) I/O unit encrypts that message for the smart gateway unit.

d) Smart gateway unit receives the encrypted message and processes it.

e) Smart gateway unit verifies the signature if the message was signed (for `SrcUnitVerified` header in step l).

f) Smart gateway unit selects the designated controlled equipment via the `DestGateway` header.

g) Smart gateway unit extracts the username and password from the decrypted message.

h) Smart gateway unit constructs “client-first” SCRAM string.

i) Smart gateway unit initiates the user authentication via equipment gateway API.

j) Controlled equipment computes “server-first” SCRAM string and returns it together with one of the supported hashing algorithms as response to the smart gateway unit.

k) Smart gateway unit computes the “client-final” SCRAM string.

l) Smart gateway unit transforms the message.

m) Smart gateway unit sends controlled equipment message and computed “client-final” SCRAM string to the controlled equipment.

n) Controlled equipment validates received “client-final” SCRAM string.

o) Controlled equipment performs logic for the received message.

p) If the logic results in a response, the controlled equipment either produces a response message and returns it synchronously or asynchronously to the smart gateway unit.

q) Smart gateway unit transforms the response message.

r) Smart gateway unit encrypts the response message for the I/O unit before sending it to the I/O unit.

s) I/O unit receives the encrypted message and processes it.

While the previous steps showcase the success-path, the following error cases in Table A.2 shall be handled in between those steps. In each of these error cases, the 4891-smart gateway unit shall stop the transmission process to the controlled equipment for the original 4891-message and instead create an error response message (type `base.error`, see 6.11.2) containing an error code as specified in Table A.2 and with these properties:

— set `SrcGateway` to identifier of controlled equipment;

— set `ReferenceId` to the id of the original message;

— set contained error code to the code specified in Table A.2 related to the error case;

— encrypt the message before sending it to the 4891-I/O unit.

Table A.2 — Overview of error cases in validation logic performed on controlled equipment

Related step	Validation logic	Error code
i)	If controlled equipment does not find the provided user, then an error with HTTP- status `401` shall be returned.	`AUTH_FAILURE`
m)	If the received "client-final" SCRAM string was invalid, then an error with HTTP-status `401` shall be returned.	`AUTH_FAILURE`
m)	If the received message requires user authentication (AOP), but no "client-final" SCRAM string was provided with it from the 4891-smart gateway unit, then an error with HTTP-status `401` shall be returned.	`AUTH_MISSING`
n)	If SCRAM-authentication was successful but the user does not have the required role/permission (authorization) for the operation, then an error with HTTP-status `403` shall be returned.	`AUTH_FORBIDDEN`

Figures A.7 to A.15 illustrate the expected communication behaviour between 4891-I/O unit, 4891-smart gateway unit and controlled equipment in different situations, when exchanging 4891-messages between 4891-I/O unit and controlled equipment.

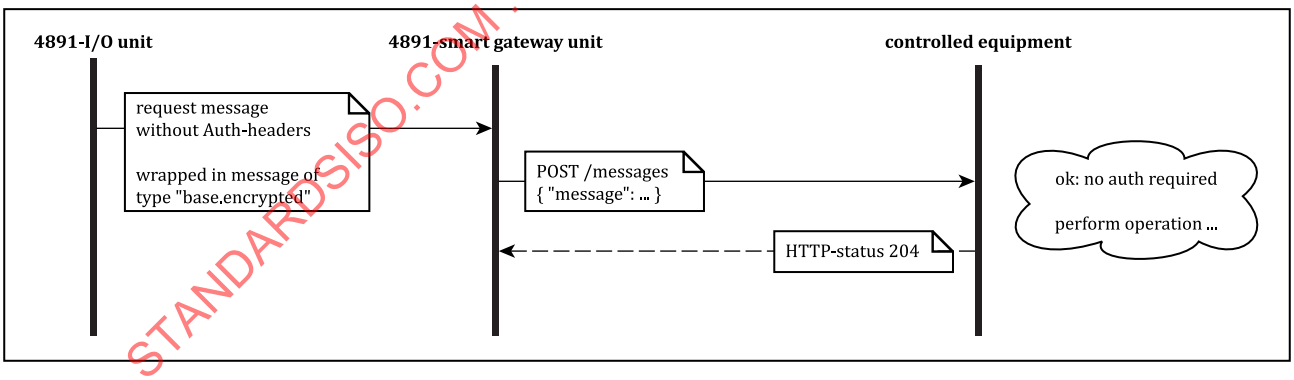


Figure A.7 — Messaging with controlled equipment — Non-AOP without response

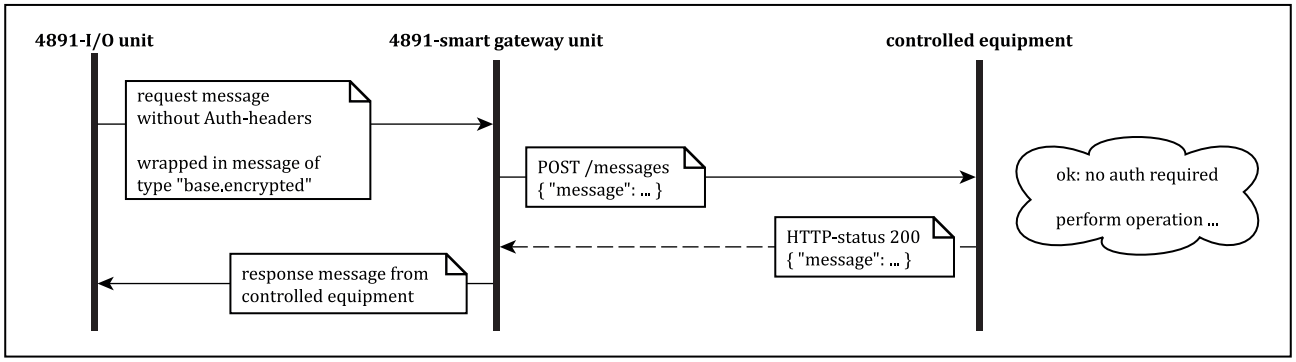


Figure A.8 — Messaging with controlled equipment — Non-AOP with response

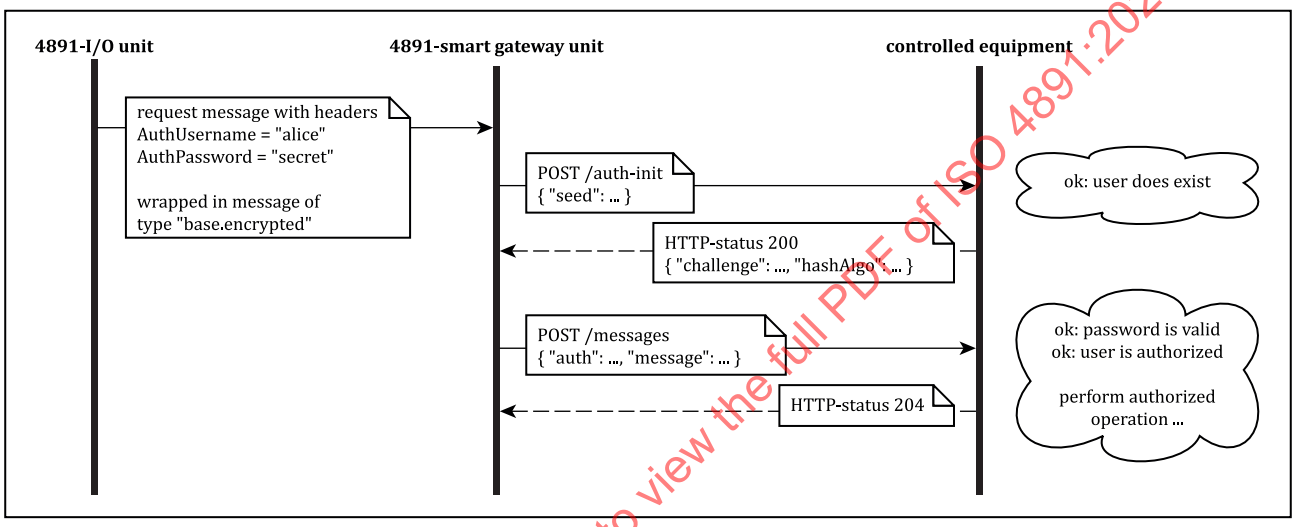


Figure A.9 — Messaging with controlled equipment — AOP without response

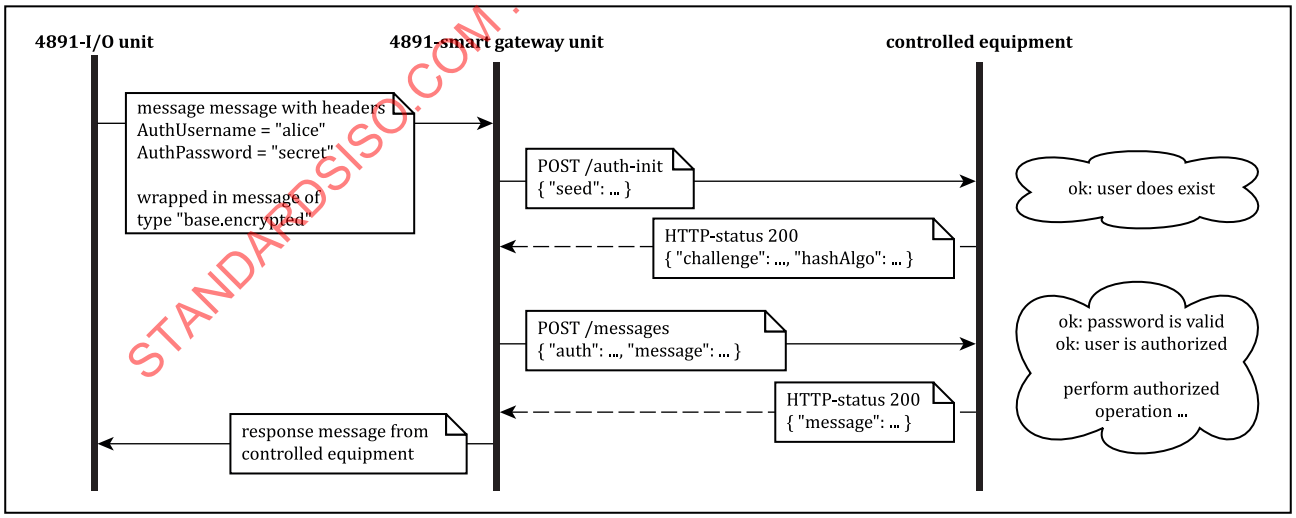


Figure A.10 — Messaging with controlled equipment — AOP with response

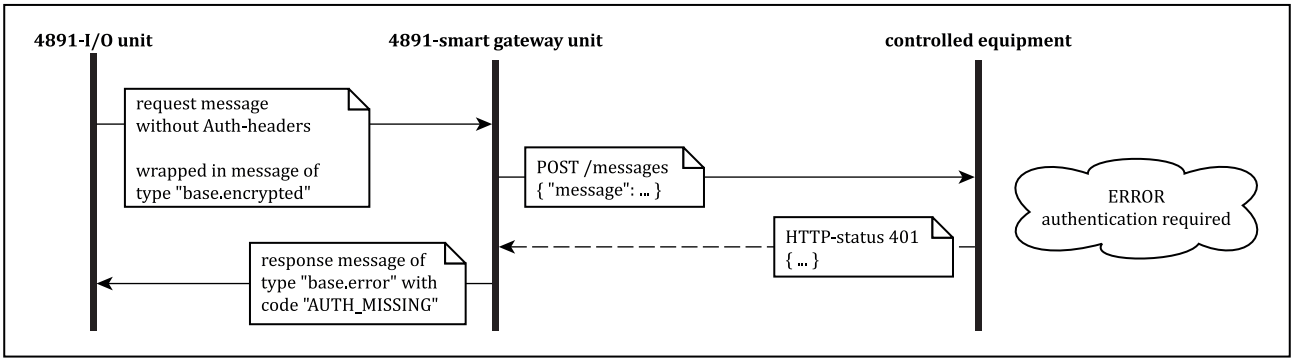


Figure A.11 — Messaging with controlled equipment — AOP failure “missing credentials”

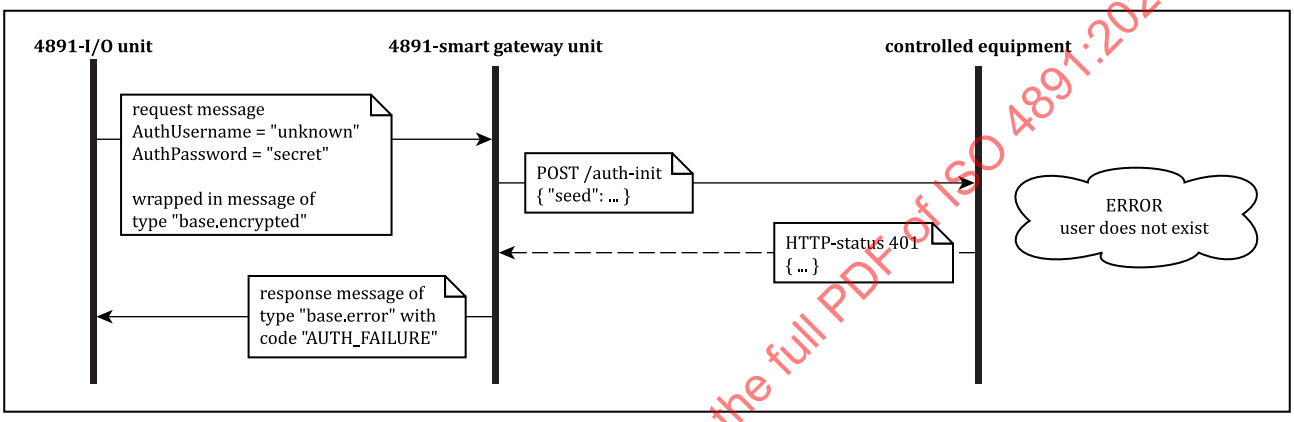


Figure A.12 — Messaging with controlled equipment — AOP failure “invalid username”

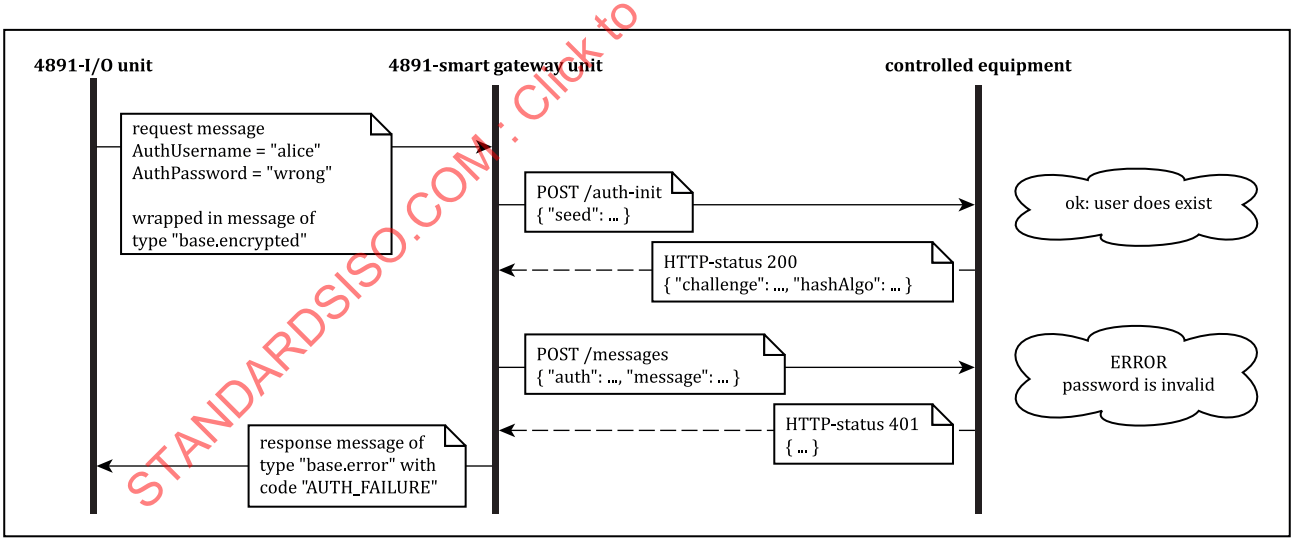


Figure A.13 — Messaging with controlled equipment — AOP failure “invalid password”

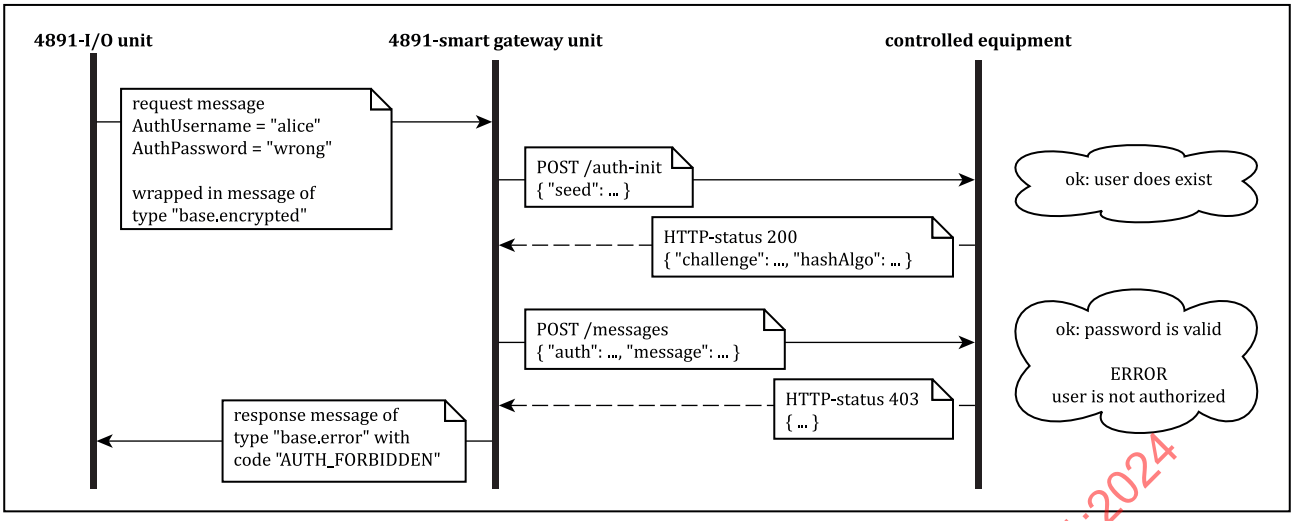


Figure A.14 — Messaging with controlled equipment — AOP failure "forbidden"

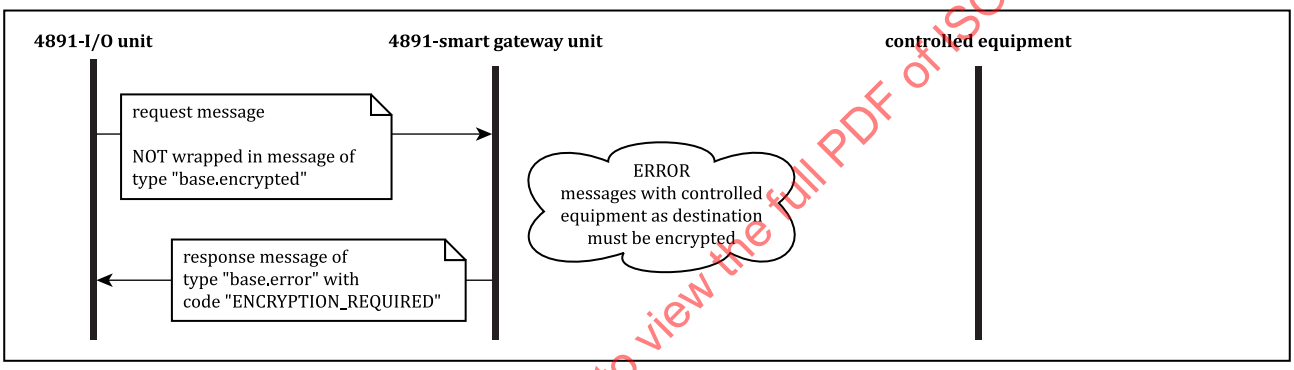


Figure A.15 — Messaging with controlled equipment — General failure "not encrypted"

Figure A.16 illustrates a full example for creating a 4891-message on a 4891-I/O unit and sending it to some controlled equipment via 4891-smart gateway unit. This example also showcases the correct wrapping for signing and encrypting the message on the source 4891-I/O unit, so that it can be accepted on the 4891-smart gateway unit and delivered as "source unit verified" to the controlled equipment.

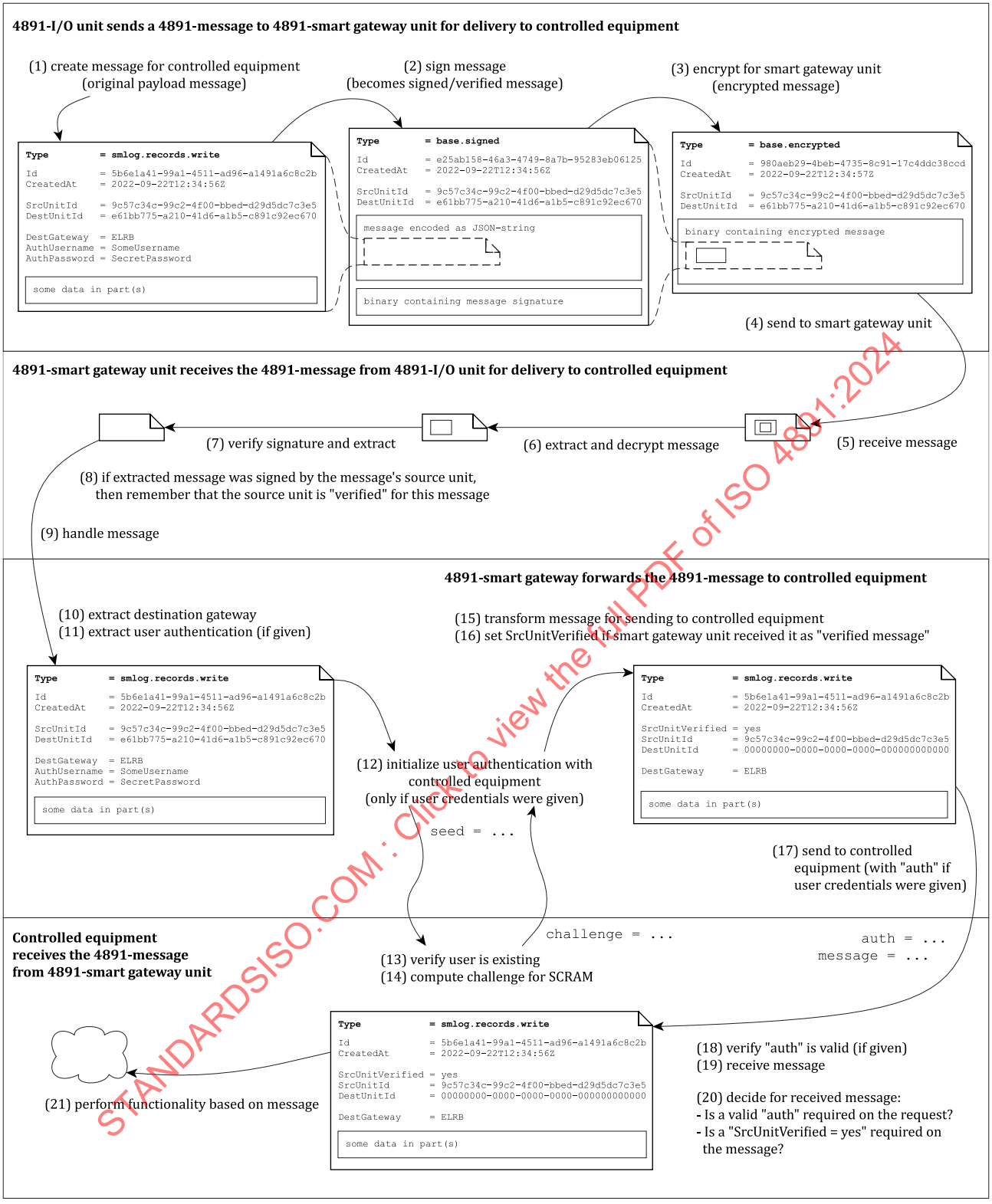


Figure A.16 — Sending a 4891-message from a 4891-I/O unit to controlled equipment

A.7 Equipment gateway API

A.7.1 General

The controlled equipment shall operate a HTTP-server implementing the following API endpoints specified in A.7.3 to A.7.4.

The 4891-smart gateway unit shall use these endpoints to initiate data exchanges with the controlled equipment. The base URL of this HTTP-API (`EquipmentGatewayApiBaseUrl`) shall be configurable in the 4891-smart gateway unit as instructed by the manufacturer's documentation.

A.7.2 Salted challenge response authentication mechanism

The authentication method called the Salted Challenge Response Authentication Mechanism (SCRAM) [5] shall be used between the 4891-smart gateway unit and controlled equipment for AOP. By doing so, the 4891-smart gateway unit proves the knowledge of a user's credentials to the controlled equipment without needing to transmit the user's password in cleartext to the controlled equipment.

In its full form, SCRAM defines four values ("client-first", "server-first", "client-final" and "server-final") that allow bidirectional authentication. Since the controlled equipment is part of the secured network, it shall be trusted by default. Therefore, only the first three of these values are required to prove the user's authentication (via 4891-smart gateway unit) to the controlled equipment. The three values are transmitted in two HTTP-requests, one for initializing and one for transmitting the message:

— `client-first`: sent to controlled equipment in the initial request (see A.7.3);

— `server-first`: computed by controlled equipment as result of initial request;

— `client-final`: sent to controlled equipment together with message (see A.7.4).

To make use of SCRAM, the controlled equipment shall hash the user's password and store the hashed result in its database. One of the following hashing algorithms shall be used by the controlled equipment:

— SHA-256

— SHA-384

— SHA-512

The 4891-smart gateway unit shall provide support for all three of those hash algorithms.

A.7.3 Endpoint "InitiateUserAuthentication"

This endpoint allows initiating the SCRAM process to authenticate a username and password pair known by the 4891-smart gateway unit via the controlled equipment.

Table A.3 specifies the signature of this endpoint.

Table A.3 — HTTP-signature of endpoint "InitiateUserAuthentication"

Method	POST	
Path	<EquipmentGatewayApiBaseUrl>/auth-init	

Request	**Content**	{
		"seed": <scram-client-first-string>
		}

Response	(A) provided username (in seed) was found and is applicable for authentication	
	Status	200

	Content	{
		"challenge": <scram-server-first-string>
		"hashAlgo": "SHA256" | "SHA384" | "SHA512"
		}

	(B) provided username (in seed) was not found or is not applicable for authentication	
	Status	403

The 4891-smart gateway unit shall generate a pseudo-random "nonce" value and combine it with the username to authenticate. That value shall be encoded as "client-first" string (see Reference [5]) and placed in the "seed" field of the request.

If the provided username (in "client-first" value) is accepted for authentication, then the controlled equipment shall respond with a "server-first" message string (see Reference [5]) and the hashing algorithm to use as part of SCRAM. The data used to build these values shall be sourced from the controlled equipment's database. Additionally, the hashing algorithm to be used as part of SCRAM shall be selected by the controlled equipment, based on the algorithm that has been used for the password hash stored in the controlled equipment's database.

If the provided username (in "client-first" string) is not acceptable for authentication, or if no such user exists in the controlled equipment's database, then the controlled equipment shall respond with an HTTP-error code of 403.

A.7.4 Endpoint "SendMessage"

This endpoint allows sending a message to the controlled equipment. There are two variants of doing so:

a) an authenticated message, requiring a SCRAM "client-final-string" (see A.7.2), utilized for AOP mode messages;

b) an un-authenticated message.

The 4891-smart gateway unit shall pick the appropriate variant based on the processing logic described in A.6.5.

Table A.4 specifies the signature of this endpoint.

Table A.4 — HTTP-signature of endpoint "SendMessage"

Method	`POST`	
Path	`<EquipmentGatewayApiBaseUrl>/messages`	
Request	(A) If the message requires user authentication (i.e. requires AOP mode, see A.5).	

	Content	`{`
		`  "auth": <scram-client-final-string>`
		`  "message": <message>`
		`}`

	(B) If the message does not require user authentication i.e. does not required AOP mode)	

	Content	`{`
		`  "message": <message>`
		`}`

Response	(A) If the received message did not produce response message, or (B) if the response message will be computed and send back asynchronously via smart gateway API (see A.8.2).	
	Status	`204` (No Content)
	(C) If the response message has been computed synchronously as part of the HTTP-response.	
	Status	`200`

	Content	`{`
		`  "message": <message>`
		`}`

	(D) If the message required user authentication (AOP), but none was provided (i.e. request was of case B), or (E) if the provided authentication was invalid.	
	Status	`403`

A.8 Smart gateway API

A.8.1 General

The 4891-smart gateway unit shall operate a HTTP-server implementing the API endpoint specified in A.8.2.

The controlled equipment may use these endpoints to send messages to the 4891-smart gateway unit. The base URL of this HTTP-API (`SmartGatewayApiBaseUrl`) shall be configurable in the controlled equipment as instructed by manufacturer's documentation. If the controlled equipment does not use these API endpoints, then that configuration shall be optional.

A.8.2 Endpoint "SendMessage"

This endpoint allows sending a message to the 4891-smart gateway unit.

Table A.5 specifies the signature of this endpoint.

Table A.5 — HTTP-signature of endpoint "SendMessage"

Method	`POST`	
Path	`<SmartGatewayApiBaseUrl>/messages`	

Request	**Content**	`{`
		`  "message": <message>`
		`}`

Response	**Status**	`204` (No Content)

A.9 Additional standard message headers

Table A.6 lists additional standard message headers for 4891-messages (see 5.3.3.2). These headers are used on messages transmitted between 4891-units and controlled equipment.

Table A.6 — Additional standard 4891-message headers

Header name	Type	Description
`SrcGateway`	`<string>`	Placed on messages by the controlled equipment for messages received from controlled equipment. Defines the gateway (i.e. the controlled equipment type) that the message originated from (e.g. entered the smart application network).
`DestGateway`	`<string>`	Placed on messages by 4891-units for messages to be communicated to controlled equipment. Defines the gateway (i.e. the controlled equipment type) that the message shall be delivered to through the smart gateway unit.
`AuthUsername`	`<string>`	Authentication credentials of the user operating the 4891-unit that created the message with these headers. These user credentials are specific to the controlled equipment that is addressed by the message containing these headers.
`AuthPassword`	`<string>`	
`SrcUnitVerified`	`<string>`	Is placed on messages by the smart gateway unit, when sent to controlled equipment. A value of `yes` states that the smart gateway unit did verify that the message was created by the stated source unit in `SrcUnitId`, i.e. that the 4891-smart gateway unit has validated a signature.

4891-I/O units shall provide the header `DestGateway` for messages destined to controlled equipment when sending them to the 4891-smart gateway unit. The header `DestGateway` shall only be used on messages where the header `DestUnitId` is set to the smart gateway unit's identifier.

The controlled equipment shall provide the header `SrcGateway` for messages sent to the 4891-smart gateway unit.

Valid values for the headers `DestGateway` and `SrcGateway` shall be taken from the set of available gateways supported by the 4891-smart gateway unit in the network. Other I/O units shall query those available gateways (see A.4.4) from the smart gateway unit to lookup the availability of gateways to controlled equipment connected to the smart gateway unit.

The header `AuthUsername` and `AuthPassword` shall only be used on messages where the header `DestGateway` is set. The header `AuthPassword` shall only be used on messages where the header `AuthUsername` is set.

The headers `AuthUsername` and `AuthPassword` shall be specified by 4891-I/O units for messages that are intended to be delivered to the controlled equipment (identified by the `DestGateway` header in the message). Certain message types specific to the controlled equipment can override this requirement. Such message types are explicitly stated to do so in their definition.

A.10 4891-message functionality

A.10.1 General

This clause defines the basic 4891-messages that shall be supported by the 4891-smart gateway unit and 4891-I/O units that are destined to interact with smart gateway units in the context of controlled equipment use cases.

A.10.2 Operation “GetEquipmentGateways”

A.10.2.1 General

This operation is intended to be used by 4891-I/O units to lookup the available controlled equipment that is connected to the smart gateway unit. With the returned information, those I/O units know what equipment can be addressed via the smart gateway unit. In case those I/O units support the message types

to communicate with the equipment, they can start interacting with the equipment by exchanging messages of those types.

The 4891-smart gateway shall react to messages of type `smgw.gateways.query` by returning the gateway identifiers of connected equipment in the `gateways` list of the response message.

A.10.2.2 Message type "smgw.gateways.query"

Table A.7 specifies the composition for this message type.

Table A.7 — Composition of 4891-message type "smgw.gateways.query"

Message type	`smgw.gateways.query`
Data parts	none

A.10.2.3 Message type "smgw.gateways.queryresp"

Table A.8 specifies the composition for this message type.

Table A.8 — Composition of 4891-message type "smgw.gateways.queryresp"

Message type	`smgw.gateways.queryresp`				
Message header	**Field**		**Field type**	**Required**	**Description**
	`ReferenceId`		`<uuid>`	true	The message id of the answered `smgw.gateways.query` message.
Data part 1	**Type**	`Record`			
	Data	**Field**	**Field type**	**Required**	**Description**
		`gateways`	`<list of string>`	true	The equipment names of the connected controlled equipment which can be addressed via the smart gateway unit. These names shall be documented together with equipment specific message types.

A.10.3 Operation "ValidateUserCredentials"

A.10.3.1 General

This operation is intended to be used by 4891-I/O units to validate the credentials of an authorized user with the controlled equipment.

With regard to successful authentication, the controlled equipment shall respond with a message of type `smgw.users.authresp`. The data part of that message shall be used for information about the authenticated user as far as applicable.

Authentication failure is handled by the 4891-smart gateway unit mediating the communication between controlled equipment and requesting I/O unit (see Figures A.7 to A.15, and related authentication logic in A.7.2).

A.10.3.2 Message type "smgw.users.auth"

This message type shall be sent to the controlled equipment as part of an authorized operation (see A.5).

Table A.9 specifies the composition for this message type.

Table A.9 — Composition of 4891-message type "smgw.users.auth"

Message type	`smgw.users.auth`