



BSI Standards Publication

Ships and marine technology — Interoperability of smart applications for ships

National foreword

This British Standard is the UK implementation of ISO 4891:2024.

The UK participation in its preparation was entrusted to Technical Committee SME/32, Ships and marine technology - Steering committee.

A list of organizations represented on this committee can be obtained on request to its committee manager.

Contractual and legal considerations

This publication has been prepared in good faith, however no representation, warranty, assurance or undertaking (express or implied) is or will be made, and no responsibility or liability is or will be accepted by BSI in relation to the adequacy, accuracy, completeness or reasonableness of this publication. All and any such responsibility and liability is expressly disclaimed to the full extent permitted by the law.

This publication is provided as is, and is to be used at the recipient's own risk.

The recipient is advised to consider seeking professional guidance with respect to its use of this publication.

This publication is not intended to constitute a contract. Users are responsible for its correct application.

© The British Standards Institution 2024
Published by BSI Standards Limited 2024

ISBN 978 0 539 13757 6

ICS 35.240.60; 47.020.99

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 30 November 2024.

Amendments/corrigenda issued since publication

Date	Text affected
------	---------------



**International
Standard**

ISO 4891

**Ships and marine technology —
Interoperability of smart
applications for ships**

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

**First edition
2024-11**

Reference number
ISO 4891:2024(en)

Full document details: BS ISO 4891:2024

© ISO 2024  **Documents**
Document Sharing Hub



COPYRIGHT PROTECTED DOCUMENT

© ISO 2024

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

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	vi
Introduction	vii
1 Scope	1
2 Normative references	1
3 Terms and definitions	1
4 Abbreviated terms	4
5 Smart application network	5
5.1 Overview.....	5
5.2 4891-components.....	8
5.2.1 General.....	8
5.2.2 4891-message broker.....	9
5.2.3 4891-service discovery.....	9
5.2.4 4891-unit registry.....	10
5.2.5 4891-units.....	10
5.3 4891-messages.....	11
5.3.1 General.....	11
5.3.2 Message structure.....	11
5.3.3 Header values.....	12
5.3.4 Data part encoding.....	13
5.3.5 Message type.....	14
5.3.6 Standard message types.....	14
5.4 Handling of outdated messages.....	15
5.5 Direct messaging.....	15
5.6 Message relaying.....	15
5.7 Trust and encryption.....	15
6 Compatibility implementation	16
6.1 General.....	16
6.2 JSON-encoding for value types.....	16
6.2.1 General.....	16
6.2.2 Common types.....	16
6.2.3 Dictionary type.....	17
6.2.4 Message type.....	18
6.3 HTTP-APIs.....	21
6.3.1 General.....	21
6.3.2 HTTP-requests.....	21
6.3.3 HTTP-request query parameters.....	21
6.3.4 HTTP-responses.....	22
6.3.5 HTTP-error responses.....	23
6.3.6 4891-unit authentication.....	23
6.4 UDP broadcasts.....	26
6.4.1 Sending UDP broadcasts.....	26
6.4.2 Listening to UDP broadcasts.....	27
6.5 4891-message broker.....	28
6.5.1 General.....	28
6.5.2 Client authentication.....	28
6.5.3 Connecting to MQTT-server.....	28
6.5.4 Message encoding.....	28
6.5.5 Publishing a 4891-message via MQTT.....	28
6.5.6 Subscribe to 4891-messages via MQTT.....	29
6.6 4891-service discovery.....	30
6.6.1 General.....	30
6.6.2 Service connectors.....	31
6.6.3 Service discovery API clients.....	32

6.6.4	Service discovery API server.....	32
6.6.5	Service discovery API discovery packet.....	33
6.6.6	Service discovery API examples.....	33
6.7	4891-unit registry.....	34
6.7.1	General.....	34
6.7.2	Tracking of unit information.....	35
6.7.3	Unit registry API clients.....	35
6.7.4	Unit registry API server.....	35
6.7.5	Unit registry API examples.....	38
6.8	4891-unit.....	40
6.9	Direct messaging API.....	41
6.9.1	General.....	41
6.9.2	Direct messaging API clients.....	41
6.9.3	Direct messaging API server.....	42
6.9.4	Direct messaging API discovery packet.....	43
6.9.5	Direct messaging API examples.....	43
6.10	Trusted communication.....	44
6.10.1	General.....	44
6.10.2	Public key infrastructure.....	44
6.10.3	Root certificates.....	45
6.10.4	Unit certificates.....	47
6.10.5	Signing data (digital signatures).....	49
6.10.6	Encrypting data.....	50
6.11	Messaging.....	51
6.11.1	General.....	51
6.11.2	Error message.....	51
6.11.3	Message meta structure.....	52
6.11.4	Receiving message from another unit.....	53
6.11.5	General message processing logic.....	53
6.11.6	Message relaying logic.....	54
6.11.7	Message handling logic.....	55
7	Test methods.....	55
7.1	General.....	55
7.1.1	Manufacturable products.....	55
7.1.2	Testing and classification.....	57
7.1.3	Use of simulated equipment.....	58
7.1.4	Testing of UDP broadcasts.....	59
7.1.5	Testing of HTTP-API servers.....	59
7.1.6	Testing of HTTP-API clients.....	59
7.1.7	Inspecting 4891-messages exchanged between 4891-units.....	60
7.2	4891-compliant equipment tests.....	60
7.2.1	General.....	60
7.2.2	4891-message broker tests.....	60
7.2.3	4891-service discovery tests.....	62
7.2.4	4891-unit registry tests.....	63
7.2.5	4891-smart gateway unit tests.....	66
7.2.6	4891-I/O unit tests.....	67
7.3	Shared functionality tests.....	67
7.3.1	General.....	67
7.3.2	4891-component tests.....	68
7.3.3	4891-unit tests.....	71
7.3.4	Message broker client tests.....	75
7.3.5	Service discovery API client tests.....	76
7.3.6	Unit registry API client tests.....	77
7.3.7	Direct messaging API client tests.....	78
7.3.8	Message relaying tests.....	78
7.3.9	Root certificate properties tests.....	79
7.3.10	Unit certificate properties tests.....	79

7.3.11 4891-message properties tests 80

7.3.12 UDP discovery broadcast sending tests..... 81

7.3.13 UDP discovery broadcast listening tests 81

7.3.14 HTTP-API server tests..... 82

7.3.15 HTTP-API client tests 83

Annex A (normative) Smart gateway — Interface to controlled equipment 85

Annex B (normative) Smart logbook — Integration with ELRB..... 121

Bibliography 130

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 initiated. This digitalization has nevertheless been lagging in terms of the need for fast and reliable data collection and utilization.

In 2017, the exchange was widened to cover demands from 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/O 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.

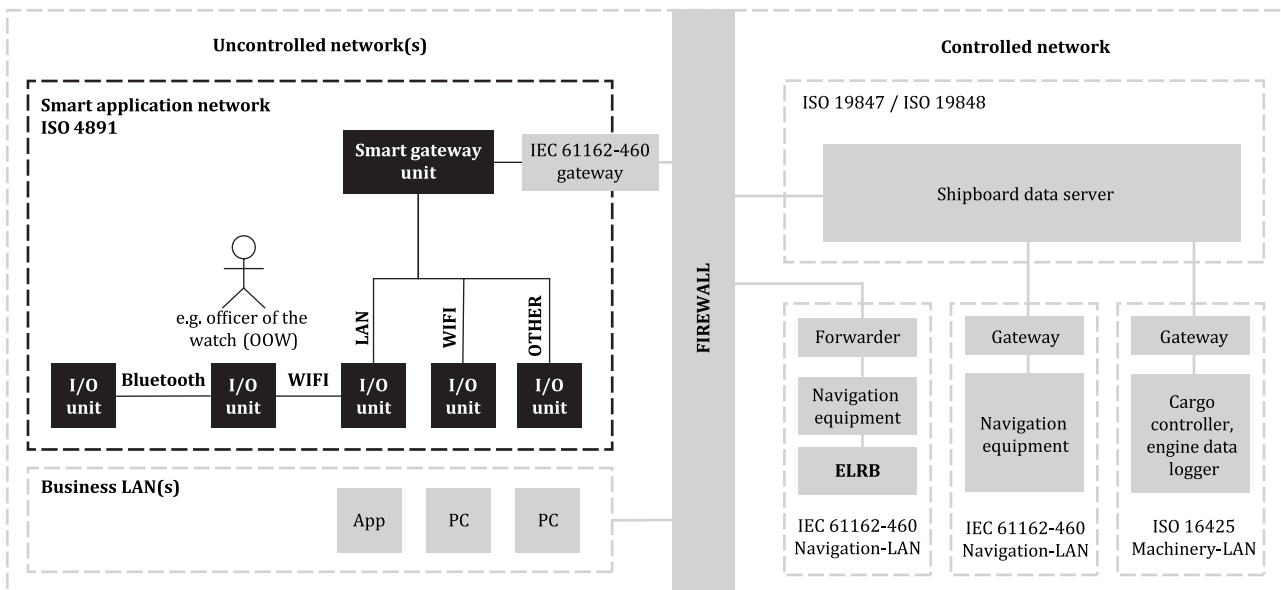
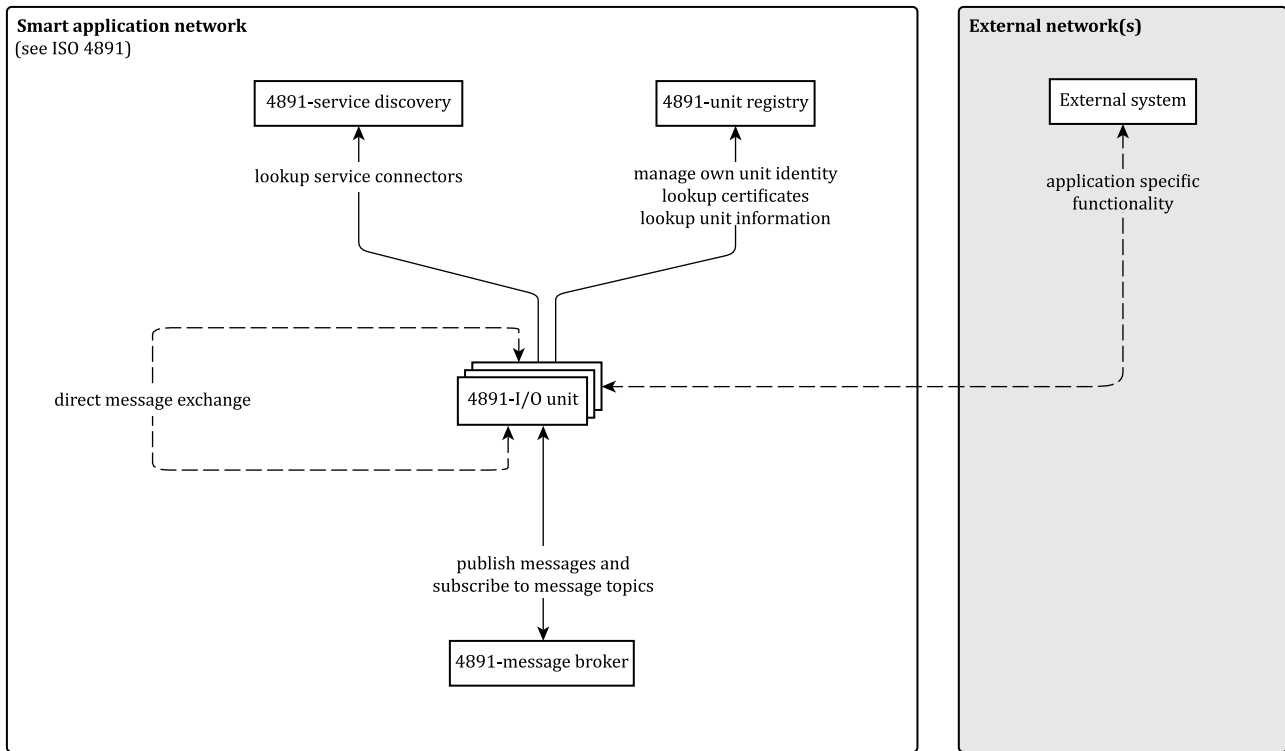


Figure 1 — Connectivity of 4891-network with other existing vessel networks

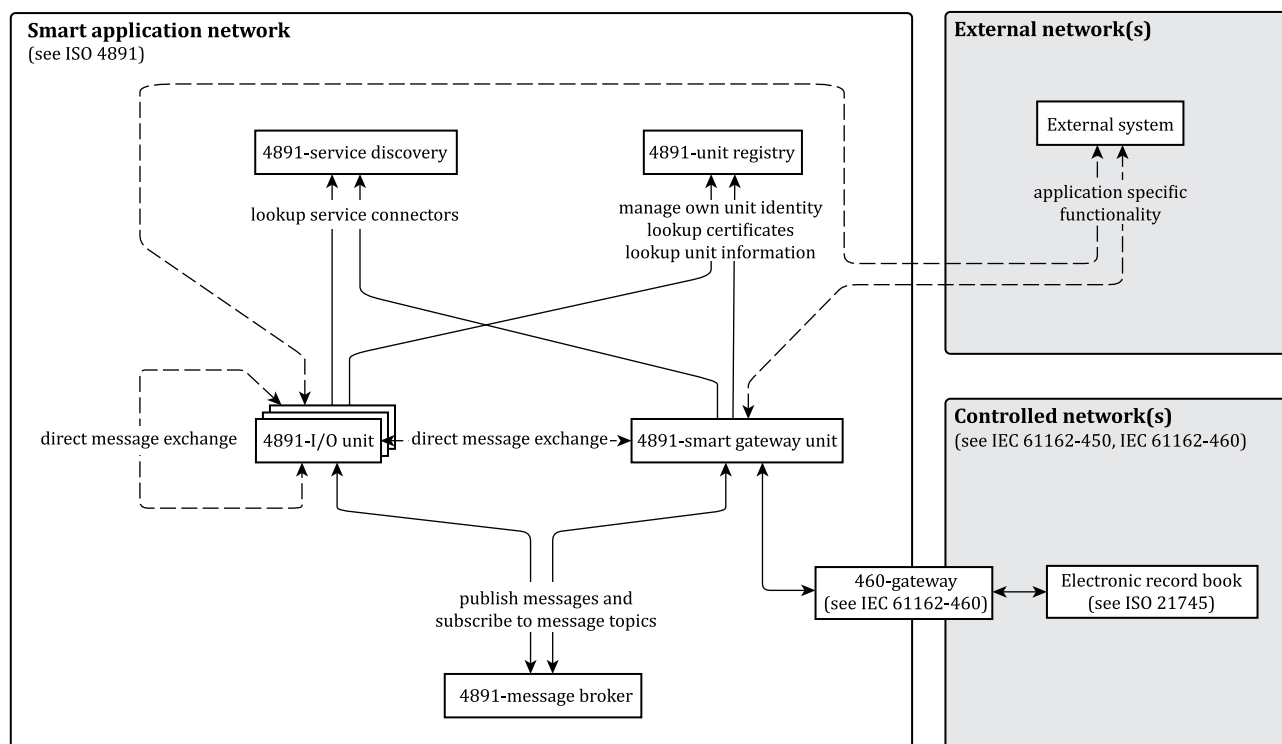


Key

- > 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

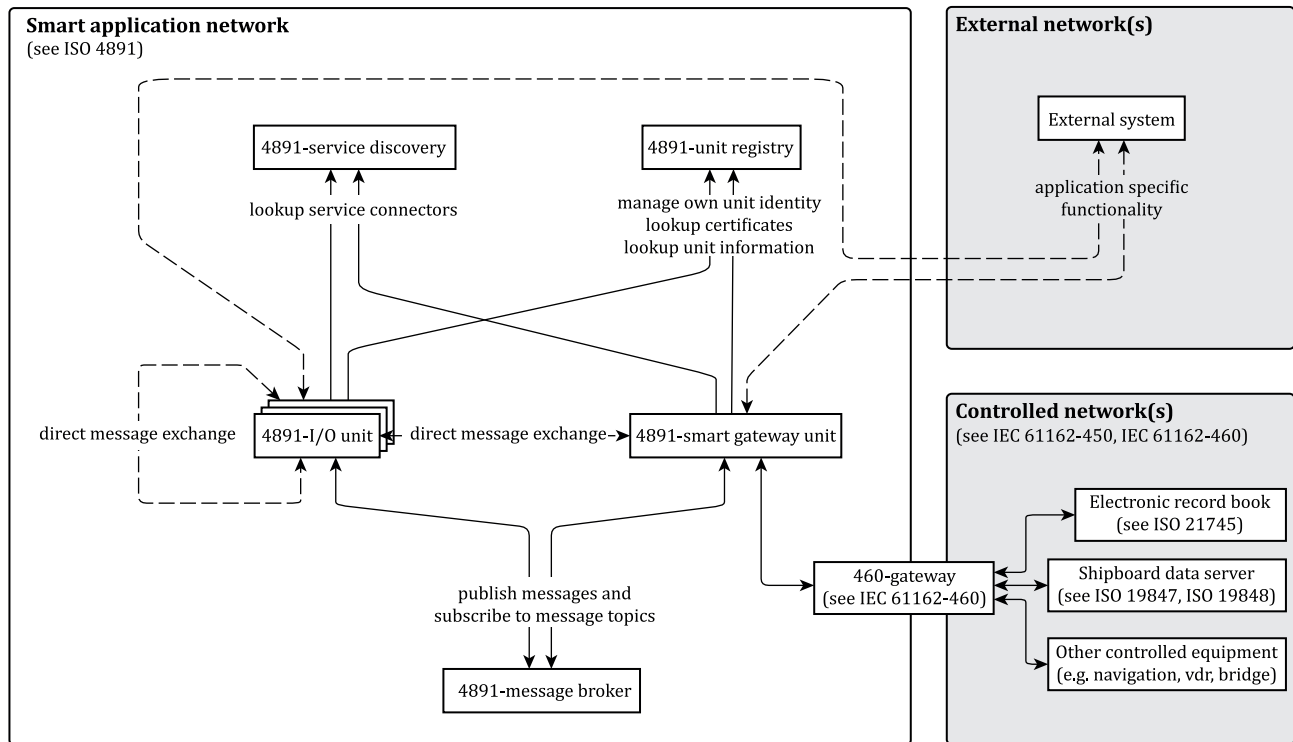


Figure 4 — Configuration example — Smart logbook as supplement to ELRB with additional functions

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

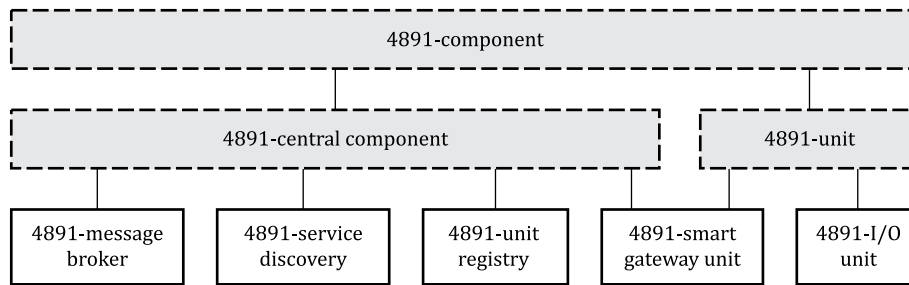


Figure 5 — Hierarchy of 4891-components

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.

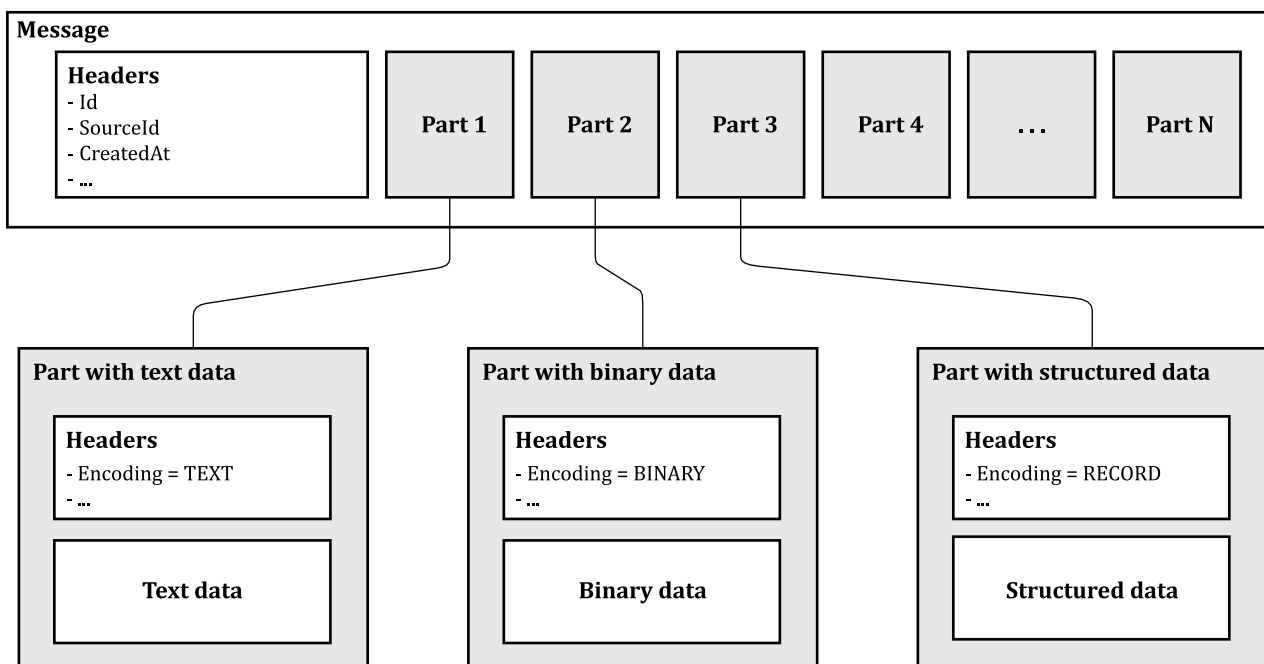


Figure 6 — Logical structure of a 4891-message

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 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 <code>true</code> or <code>false</code> .
<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: "aGVyZSBzb2llIGRhdGE = " (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 <code>/</code> -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>”

Type name	JSON type	Extended description	
<dictionary>	object	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.	
<dictionary-field-value>	mixed	Depending on the dictionary field's type this type maps to one of these types:	
		Field Type	Mapped Type
		null value	<null>
		string value	<string>
		integer value	<integer>
		boolean value	<boolean>
		list value	<list of dictionary-field-value>
		dictionary value	<dictionary>

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>”

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

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": "SSBhbSBzb2llIGJpbmFyeSBjb250ZW50Lg=="
    },
    {
      "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-daf7c238f269\\",\\"DestUnitId\\":\\"58675eb7-a9ea-4b7f-b80b-b0ebb8209016\\",\\"CreatedAt\\":\\"2021-03-03T12:45:56Z\\"},\\"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}}}}]]]"
    },
    {
      "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):

`aGVyZSBzb211IGRhGE=`

Query-encoding:

`aGVyZSBzb211IGRhGE%3D`

Used as value on an URL parameter:

`http://127.0.0.1?someParam=aGVyZSBzb211IGRhGE%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:

- Lookup the unit certificate for the unit identifier provided in the request.
- Validate that the signature in the request relates to the unit certificate.
- Validate that the signature is valid for the data field of the request for that unit certificate.
- 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 = \"dGhlLXNpZ25hdHVyZS1ieXRlcw==\"
```

The data variable is JSON-decoded for further processing:

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

The following variables are extracted from the dataValue variable:

```
unitId = \"2b985ef2-e305-4fae-ad5f-20630c9109a0\"
authToken = \"d7cbb91f6367ee5a73c25999d91c1\"
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 packet 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.

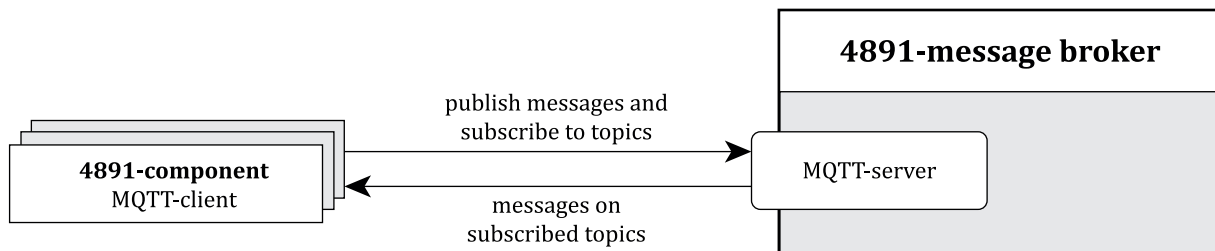


Figure 7 — Behaviour interfaces of 4891-message broker

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

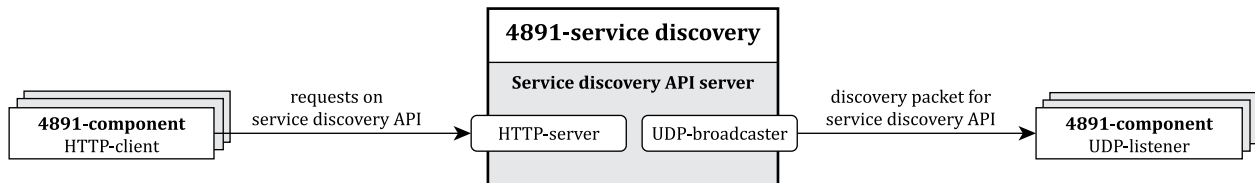


Figure 8 — Behaviour interfaces of 4891-service discovery

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	<code><string></code>	Yes	Host of the MQTT-server e.g. "192.168.10.18"
mqttPort	<code><integer></code>	Yes	Port number of the MQTT-server e.g. 1883
mqttUser	<code><string></code>	No	Username that units shall use when connecting to the MQTT-server. If empty or missing, then anonymous access is used instead.
mqttPassword	<code><string></code>	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 <code>mqttUser</code> 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	<code><base-url></code>	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 possible 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	<pre>{ "messageBroker": <null> <message-broker-connector> "unitRegistry": <null> <unit-registry-connector> "smartGatewayUnit": <null> <smart-gateway-unit-connector> }</pre>

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: