
Version 2.5

OMG Document Number: formal/2022-04-01

Standard document URL: https://www.omg.org/spec/DDSI-RTPS

Machine Consumable Files: https://www.omg.org/spec/DDSI-RTPS/20210212

Normative:

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a
license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.

IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 9C Medway ROAD, PMB 274, Milford, MA 01757, U.S.A.

TRADEMARKS

CORBA®, CORBA logos®, FIBO®, Financial Industry Business Ontology®, FINANCIAL INSTRUMENT GLOBAL IDENTIFIER®, IIOP®, IMM®, Model Driven Architecture®, MDA®, Object Management Group®, OMG®, OMG Logo®, SoaML®, SOAML®, SysML®, UAF®, Unified Modeling Language®, UML®, UML Cube Logo®, VSIPL®, and XMl® are registered trademarks of the Object Management Group, Inc.

For a complete list of trademarks, see: https://www.omg.org/legal/tm_list.htm. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.
COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

OMG’s Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page https://www.omg.org, under Documents, Report a Bug/Issue.
Table of Contents

1 Scope ................................................................................................................................. 1
2 Conformance ....................................................................................................................... 1
3 Normative References ....................................................................................................... 1
4 Terms and Definitions ....................................................................................................... 1
5 Symbols ............................................................................................................................... 1
6 Additional Information ..................................................................................................... 3
   6.1 Changes to Adopted OMG Specifications ................................................................. 3
   6.2 How to Read this Specification ................................................................................ 3
   6.3 Acknowledgments ....................................................................................................... 3
   6.4 Statement of Maturity ............................................................................................... 5
7 Overview ............................................................................................................................. 7
   7.1 Introduction ................................................................................................................ 7
   7.2 Requirements for a DDS Wire-protocol .................................................................... 7
   7.3 The RTPS Wire-protocol ......................................................................................... 8
   7.4 The RTPS Platform Independent Model (PIM) ....................................................... 8
   7.5 The RTPS Platform Specific Model (PSM) .......................................................... 11
   7.6 The RTPS Transport Model .................................................................................. 11

8 Platform Independent Model (PIM) .................................................................................. 13
   8.1 Introduction ............................................................................................................... 13
   8.2 Structure Module .................................................................................................... 13
   8.3 Messages Module .................................................................................................... 29
   8.4 Behavior Module .................................................................................................... 64
   8.5 Discovery Module .................................................................................................... 117
   8.6 Versioning and Extensibility .................................................................................. 131
   8.7 Implementing DDS QoS and advanced DDS features using RTPS ......................... 131

9 Platform Specific Model (PSM): UDP/IP ......................................................................... 143
   9.1 Introduction ............................................................................................................. 143
   9.2 Notational Conventions ....................................................................................... 143
   9.3 Mapping of the RTPS Types .................................................................................. 144
   9.4 Mapping of the RTPS Messages ............................................................................ 152
   9.5 Mapping to UDP/IP Transport Messages ............................................................. 176
   9.6 Mapping of the RTPS Protocol ............................................................................. 176

10 Serialized Payload Representation .................................................................................. 193
   10.1 Introduction ........................................................................................................... 193
   10.2 SerializedPayloadHeader and Representation Identifier ...................................... 193
   10.3 SerializedPayload for RTPS discovery built-in endpoints .................................... 194
   10.4 SerializedPayload for other RTPS built-in endpoints ........................................... 194
   10.5 SerializedPayload for user-defined DDS Topics ................................................. 194
   10.6 Example for Built-in Endpoint Data .................................................................... 195
   10.7 Example for User-defined Topic Data .................................................................. 197

A References ......................................................................................................................... 199
Preface

About the Object Management Group

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG’s specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at https://www.omg.org/.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Specifications are available from this URL: https://www.omg.org/spec/.

All of OMG’s formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
9C Medway Road, PMB 274
Milford, MA 01757, USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: pubs@omg.org

Certain OMG specifications are also available as ISO standards. Please consult http://www.iso.org

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.


Helvetica/Arial - 10 pt: Exceptions

Note – Terms that appear in italics are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to: https://issues.omg.org/issues/create-new-issue.
1 Scope

This specification defines an interoperability wire protocol for DDS. Its purpose and scope are to ensure that applications based on different vendors’ implementations of DDS can interoperate.

2 Conformance

Implementations of this specification must comply with the conformance statements listed in 8.4.2 of this specification.

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.


4 Terms and Definitions

For the purposes of this specification, the terms and definitions given in the normative references apply.

5 Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDR</td>
<td>Common Data Representation</td>
</tr>
<tr>
<td>DDS</td>
<td>Data Distribution Service</td>
</tr>
<tr>
<td>EDP</td>
<td>Endpoint Discovery Protocol</td>
</tr>
<tr>
<td>GUID</td>
<td>Globally Unique Identifier</td>
</tr>
<tr>
<td>PDP</td>
<td>Participant Discovery Protocol</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model</td>
</tr>
<tr>
<td>RTPS</td>
<td>Real-Time Publish-Subscribe</td>
</tr>
<tr>
<td>SEDP</td>
<td>Simple Endpoint Discovery Protocol</td>
</tr>
</tbody>
</table>
This page intentionally left blank.
6 Additional Information

6.1 Changes to Adopted OMG Specifications

This specification does not change any adopted OMG specifications. It forms a supplement to the OMG DDS specification (see https://www.omg.org/spec/DDS/1.4/).

6.2 How to Read this Specification

This specification defines the DDS Interoperability Protocol. Readers not familiar with DDS will benefit from first reading the DDS specification.

For a very high-level overview of RTPS (Real-Time Publish-Subscribe) and a brief description of the structure of this document, please refer to the Introduction. Subsequent clauses cover RTPS in much greater detail.

While providing both a PIM (Platform Independent Model) and a PSM (Platform Specific Model) contributed to the size of this document, this approach also enables a selective reader to easily pick the sub clauses of interest:

- Readers who are new to RTPS can start by reading the Structure and Messages Modules of the PIM. These Modules provide an overview of the RTPS protocol actors, how they relate to DDS Entities, what RTPS messages exist and how they are structured.
- Readers who would like to explore the RTPS message exchange protocol can read the first part of the Behavior Module. RTPS is a fairly flexible protocol and allows implementations to customize their behavior depending on how much 'state' they wish to keep on remote Endpoints. The first part of the Behavior Module lists the general requirements any compliant implementation of RTPS must satisfy to remain interoperable with other implementations.
- The second part of the Behavior Module defines two reference implementations. One reference implementation maintains full state on remote Endpoints, the other none. This sub clause may be of interest to readers who want a more detailed understanding of the RTPS message exchange protocol, but it could easily be skipped by first-time readers.
- Readers interested in how RTPS handles dynamic discovery of remote Endpoints are referred to the stand-alone Discovery Module.
- For readers planning on implementing RTPS or defining a new PSM, the PSM Clause contains a detailed discussion on how the RTPS PIM is mapped to the UDP/IP PSM.
- Finally, the clause on data representation defines various data representation mechanisms for use with RTPS.

6.3 Acknowledgments

The following individuals and companies submitted content that was incorporated into this specification:

- Real-Time Innovations, Inc.
- THALES
- PrismTech


- (lead) Gerardo Pardo-Castellote, Ph.D., Real-Time Innovations. gerardo.pardo AT rti.com
- Virginie Watine, THALES, virginie.watine AT thalesgroup.com
- Hans Van't Haag, PrismTech Ltd.

Revision 2.0 (2007) finalization Task Force members and participants:

- Gerardo Pardo-Castellote, Ph.D., Real-Time Innovations. gerardo.pardo AT rti.com
• Carlo Cloet, Real-Time Innovations. carlo AT rti.com
• Ken Brophy, Real-Time Innovations. ken AT rti.com
• Virginie Watine, THALES, virginie.watine AT thalesgroup.com
• Hans Van't Hag, PrismTech Ltd.
• Angelo Corsaro, Ph.D., SELEX SI. acorsaro AT selex-si.com
• Charles Fudge, NSWC Dalghren, charles.fudge AT navy.mil
• Char Wales, MITRE, charwing AT mitre.org
• Victor Giddings, Objective Interface Systems, victor.giddings AT ois.com

Revision 2.1 (2008) Task Force members and participants:
• Gerardo Pardo-Castellote, Ph.D., Real-Time Innovations. gerardo.pardo AT rti.com
• Ken Brophy, Real-Time Innovations. ken AT rti.com
• Virginie Watine, THALES, virginie.watine AT thalesgroup.com
• Hans Van't Hag, PrismTech Ltd.
• Angelo Corsaro, Ph.D., SELEX SI. angelo.corsaro AT selex.com
• Charles Fudge, NSWC Dalghren, charles.fudge AT navy.mil
• Char Wales, MITRE, charwing AT mitre.org
• Victor Giddings, Objective Interface Systems, victor.giddings AT ois.com

Revision 2.2 (2014) Task Force members and participants:
• (chair) Gerardo Pardo-Castellote, Ph.D., Real-Time Innovations. gerardo.pardo AT rti.com
• Tetsuo Kotoku, AIST, t.kotoku AT ais.go.jp
• Mathew Hause, Atego
• Ken Rode, Gallium Kongsberg, krode AT gallium.com
• Ron Townsen, General Dynamics, Ronald.Townsen AT gd-ms.com
• Char Wales, MITRE, charwing AT mitre.org
• Charles Fudge, NSWC Dalghren, charles.fudge AT navy.mil
• Adam Mitz, Object Computing Incorporated, mitza AT objectcomputing.com
• Victor Giddings, Objective Interface Systems, victor.giddings AT ois.com
• Angelo Corsaro, Ph.D., PrismTech, angelo.corsaro AT prismtech.com
• Johnny Willemsen, Remedy IT, jwillemsen AT remedy.nl
• Virginie Watine, THALES, virginie.watine AT thalesgroup.com
• Clark Tucker, Twin Oaks Computing, ctucker AT twinoakscomputing.com
• Doug Tolberg, Unisys, Doug.Tolbert AT unisys.com

Revision 2.3 (2019) Task Force members and participants:
• (chair) Gerardo Pardo-Castellote, Ph.D., Real-Time Innovations. gerardo.pardo AT rti.com
• Charles Fudge, NSWC Dalghren, charles.fudge AT navy.mil
• Virginie Watine, THALES, virginie.watine AT thalesgroup.com
• Hakim Souami, THALES, hakim.souami AT thalesgroup.com
Johnny Willemsen, Remedy IT, jwillemsen AT remedy.nl
Erik Hendriks, ADLINK Technology Ltd., erik.hendriks AT adlinktech.com
Adam Mitz, Object Computing Incorporated, mitza AT objectcomputing.com
Mathew House, PTC, mhouse AT ptc.com
Clark Tucker, Twin Oaks Computing, ctucker AT twinoakscomputing.com
Char Wales, MITRE, charwing AT mitre.org
Angelo Corsaro, Ph.D., ADLINK Technology Ltd., angelo.corsaro AT adlinktech.com
Jaime Martin-Losa, eProsima, JaimeMartin AT eprosima.com

Revision 2.4 Task Force members and participants:
This revision number was skipped

Revision 2.5 (2020) Task Force members and participants:
(chair) Gerardo Pardo-Castellote, Ph.D., Real-Time Innovations, gerardo.pardo AT rti.com
Charles Fudge, NSWC Daughren, charles.fudge AT navy.mil
Erik Hendriks, ADLINK Technology Ltd., erik.hendriks AT adlinktech.com
Adam Mitz, Object Computing Incorporated, mitza AT objectcomputing.com
Clark Tucker, Twin Oaks Computing, ctucker AT twinoakscomputing.com
Ornulf Staff, Kongsberg Defence & Aerospace, Ornulf.Staff AT kongsberg.com
Char Wales, MITRE, charwing AT mitre.org
Syltinsy Jenkins, MITRE, spjenkins AT mitre.org
Jaime Martin-Losa, eProsima, JaimeMartin AT eprosima.com
Nick Stavros, Ph.D., Jackrabbit Consulting, nick AT drstavros.com

6.4 Statement of Maturity

The protocol specified in this proposal has proven its performance and applicability to data-distribution systems. The protocol has had more than a dozen independent implementations, both commercial and open source. These products have been deployed in hundreds of thousands of applications worldwide in the 14 years since this specification was initially adopted.

The OMG has performed interoperability demonstrations among many different implementations, including the DDS/RTPS implementations from Real-Time Innovations (Connext DDS and Connext DDS Micro), ADLink (OpenSplice DDS, Vortex Cafe, and CycloneDDS), TwinOaks Computing (CoreDX DDS), Kongsberg (InterComm DDS), Object Computing Incorporated (OpenDDS), and eProsima (FastRTPS).
7 Overview

7.1 Introduction

The recently-adopted Data-Distribution Service specification defines an Application Level Interface and behavior of a Data-Distribution Service (DDS) that supports Data-Centric Publish-Subscribe (DCPS) in real-time systems. The DDS specification used a Model-Driven Architecture (MDA) approach to precisely describe the Data-Centric communications model specifically:

- How the application models the data it wishes to send and receive.
- How the application interacts with the DCPS middleware and specifies the data it wishes to send and receive as well as the quality of service (QoS) requirements.
- How data is sent and received (relative to the QoS requirements).
- How the applications access the data.
- The kinds of feedback the application gets from the state of the middleware.

The DDS specification also includes a platform specific mapping to IDL and therefore an application using DDS is able to switch among DDS implementations with only a re-compile. DDS therefore addresses "application portability."

The DDS specification does not address the protocol used by the implementation to exchange messages over transports such as TCP/UDP/IP, so different implementations of DDS will not interoperate with each other unless vendor-specific “bridges” are provided. The situation is therefore similar to that of other messaging API standards such as JMS.

With the increasing adoption of DDS in large distributed systems, it is desirable to define a standard “wire protocol” that allows DDS implementations from multiple vendors to interoperate. The desired “DDS wire protocol” should be capable of taking advantage of the QoS settings configurable by DDS to optimize its use of the underlying transport capabilities. In particular, the desired wire protocol must be capable of exploiting the multicast, best-effort, and connectionless nature of many of the DDS QoS settings.

7.2 Requirements for a DDS Wire-protocol

In network communications, as in many other fields of engineering, it is a fact that “one size does not fit all.” Engineering design is about making the right set of trade-offs, and these trade-offs must balance conflicting requirements such as generality, ease of use, richness of features, performance, memory size and usage, scalability, determinism, and robustness. These trade-offs must be made in light of the types of information flow (e.g., periodic vs. bursty, state-based vs. event-based, one-to-many vs. request-reply, best-effort vs. reliable, small data-values vs. large files, etc.), and the constraints imposed by the application and execution platforms. Consequently, many successful protocols have emerged such as HTTP, SOAP, FTP, DHCP, DCE, RTP, DCOM, and CORBA. Each of these protocols fills a niche, providing well-tuned functionality for specific purposes or application domains.

The basic communication model of DDS is one of unidirectional data exchange where the applications that publish data “push” the relevant data updates to the local caches of co-located subscribers to the data. This information flow is regulated by QoS contracts implicitly established between the DataWriters and the DataReaders. The DataWriter specifies its QoS contract at the time it declares its intent to publish data and the DataReader does it at the time it declares its intent to subscribe to data. The communication patterns typically include many-to-many style configurations. Of primary concern to applications deploying DDS technology is that the information is distributed in an efficient manner with minimal overhead. Another important requirement is the need to scale to hundreds or thousands of subscribers in a robust fault-tolerant manner.

The DDS specification prescribes the presence of a built-in discovery service that allows publishers to dynamically discover the existence of subscribers and vice-versa and performs this task continuously without the need to contact any name servers.
The DDS specification also prescribes that the implementations should not introduce any single points of failure. Consequently, protocols must not rely on centralized name servers or centralized information brokers.

The large scale, loosely-coupled, dynamic nature of applications deploying DDS and the need to adapt to emerging transports require certain flexibility on the data-definition and protocol such that each can be evolved while preserving backwards compatibility with already deployed systems.

7.3 The RTPS Wire-protocol

The Real-Time Publish Subscribe (RTPS) protocol found its roots in industrial automation and was in fact approved by the IEC as part of the Real-Time Industrial Ethernet Suite IEC-PAS-62030. It is a field proven technology that is currently deployed worldwide in thousands of industrial devices.

RTPS was specifically developed to support the unique requirements of data-distributions systems. As one of the application domains targeted by DDS, the industrial automation community defined requirements for a standard publish-subscribe wire-protocol that closely match those of DDS. As a direct result, a close synergy exists between DDS and the RTPS wire-protocol, both in terms of the underlying behavioral architecture and the features of RTPS.

The RTPS protocol is designed to be able to run over multicast and connectionless best-effort transports such as UDP/IP. The main features of the RTPS protocol include:

- Performance and quality-of-service properties to enable best-effort and reliable publish-subscribe communications for real-time applications over standard IP networks.
- Fault tolerance to allow the creation of networks without single points of failure.
- Extensibility to allow the protocol to be extended and enhanced with new services without breaking backwards compatibility and interoperability.
- Plug-and-play connectivity so that new applications and services are automatically discovered and applications can join and leave the network at any time without the need for reconfiguration.
- Configurability to allow balancing the requirements for reliability and timeliness for each data delivery.
- Modularity to allow simple devices to implement a subset of the protocol and still participate in the network.
- Scalability to enable systems to potentially scale to very large networks.
- Type-safety to prevent application programming errors from compromising the operation of remote nodes.

The above features make RTPS an excellent match for a DDS wire-protocol. Given its publish-subscribe roots, this is not a coincidence, as RTPS was specifically designed for meeting the types of requirements set forth by the DDS application domain.

This specification defines the message formats, interpretation, and usage scenarios that underlie all messages exchanged by applications that use the RTPS protocol.

7.4 The RTPS Platform Independent Model (PIM)

The RTPS protocol is described in terms of a Platform Independent Model (PIM) and a set of PSMs.

The RTPS PIM contains four modules: Structure, Messages, Behavior, and Discovery. The Structure module defines the communication endpoints. The Messages module defines the set of messages that those endpoints can exchange. The Behavior module defines sets of legal interactions (message exchanges) and how they affect the state of the communication endpoints. In other words, the Structure module defines the protocol “actors,” the Messages module the set of “grammatical symbols,” and the Behavior module the legal grammar and semantics of the different conversations. The Discovery module defines how entities are automatically discovered and configured.
In the PIM, the messages are defined in terms of their semantic content. This PIM can then be mapped to various Platform-Specific Models (PSMs) such as plain UDP or CORBA-events.

### 7.4.1 The Structure Module

Given its publish-subscribe roots, RTPS maps naturally to many DDS concepts. This specification uses many of the same core entities used in the DDS specification. As illustrated in Figure 7.2, all RTPS entities are associated with an RTPS domain, which represents a separate communication plane that contains a set of Participants. A Participant contains Groups which contain local Endpoints. There are two kinds of endpoints: Readers and Writers. Readers and Writers are the actors that communicate information by sending RTPS messages. Writers inform of the presence and send locally available data on the Domain to the Readers which can request and acknowledge the data.

The Actors in the RTPS Protocol are in one-to-one correspondence with the DDS Entities that are the reason for the communication to occur. This is illustrated in Figure 7.3.
The Structure module is described in 8.2.

### 7.4.2 The Messages Module

The messages module defines the content of the atomic information exchanges between RTPS Writers and Readers. Messages are composed of a Header followed by a number of Submessages, as illustrated in Figure 7.4. Each Submessage is built from a series of Submessage elements. This structure is chosen to allow the vocabulary of Submessages and the composition of each Submessage to be extended while maintaining backward compatibility. The **HeaderExtension** is a special Submessage that may optionally appear immediately following the **Header**.

The Messages module is discussed at length in 8.3.

### 7.4.3 The Behavior Module

The Behavior module describes the allowed sequences of messages that can be exchanged between RTPS Writers and Readers as well as the timings and changes in the state of the Writer and the Reader caused by each message.
The required behavior for interoperability is described in terms of a minimum set of rules that an implementation must follow in order to be interoperable. Actual implementations may exhibit different behavior beyond these minimum requirements, depending on how they wish to trade-off scalability, memory requirements, and bandwidth usage.

To illustrate this concept, the Behavior module defines two reference implementations. One reference implementation is based on StatefulWriters and StatefulReaders, the other on StatelessWriters and StatelessReaders, as illustrated in Figure 7.2 - RTPS Structure Module. Both reference implementations satisfy the minimum requirements for interoperability, and are therefore interoperable, but exhibit slightly different behavior due to the difference in information they store on matching remote entities. The behavior of an actual implementation of the RTPS protocol may be an exact match or a combination of that of the reference implementations.

The Behavior module is described in 8.4.

### 7.4.4 The Discovery Module

The Discovery module describes the protocol that enables Participants to obtain information about the existence and attributes of all the other Participants and Endpoints in the Domain. This metatraffic enables every Participant to obtain a complete picture of all Participants, Readers and Writers in the Domain and configure the local Writers to communicate with the remote Readers and the local Readers to communicate with the remote Writers.

Discovery is a separate module. The unique needs of Discovery, namely the transparent plug-and-play dissemination of all the information needed to associate matching Writers and Readers make it unlikely that a single architecture or protocol can fulfill the extremely variable scalability, performance, and embeddability needs of the various heterogeneous networks where DDS will be deployed. Henceforth, it makes sense to introduce several discovery mechanisms ranging from the simple and efficient (but not very scalable), to a more complex hierarchical (but more scalable) mechanism.

The Discovery module is described in 8.5.

### 7.5 The RTPS Platform Specific Model (PSM)

A Platform Specific Model maps the RTPS PIM to a specific underlying platform. It defines the precise representation in bits and bytes of all RTPS Types and Messages and any other information specific to the platform.

Multiple PSMs may be supported, but all implementations of DDS must at least implement the PSM on top of UDP/IP, which is presented in Clause 9.

### 7.6 The RTPS Transport Model

RTPS supports a wide variety of transports and transport QoS. The protocol is designed to be able to run on multicast and best-effort transports, such as UDP/IP and requires only very simple services from the transport. In fact, it is sufficient that the transport offers a connectionless service capable of sending packets best-effort. That is, the transport need not guarantee each packet will reach its destination or that packets are delivered in-order. Where required, RTPS implements reliability in the transfer of data and state above the transport interface. This does not preclude RTPS from being implemented on top of a reliable transport. It simply makes it possible to support a wider range of transports.

If available, RTPS can also take advantage of the multicast capabilities of the transport mechanism, where one message from a sender can reach multiple receivers. RTPS is designed to promote determinism of the underlying communication mechanism. The protocol provides an open trade-off between determinism and reliability.
The general requirements RTPS poses on the underlying transport can be summarized as follows:

- The transport has a generalized notion of a unicast address (shall fit within 16 bytes).
- The transport has a generalized notion of a port (shall fit within 4 bytes), e.g., could be a UDP port, an offset in a shared memory segment, etc.
- The transport can send a datagram (uninterpreted sequence of octets) to a specific address/port.
- The transport can receive a datagram at a specific address/port.
- The transport will drop messages if incomplete or corrupted during transfer (i.e., RTPS assumes messages are complete and not corrupted).
8 Platform Independent Model (PIM)

8.1 Introduction

This clause defines the Platform Independent Model (PIM) for the RTPS protocol. Subsequent clauses map the PIM to a variety of platforms, the most fundamental one being native UDP packets.

The PIM describes the protocol in terms of a “virtual machine.” The structure of the virtual machine is built from the classes described in 8.2, which include Writer and Reader endpoints. These endpoints communicate using the messages described in 8.3. Sub clause 8.4 describes the behavior of the virtual machine, i.e., what message exchanges take place between the endpoints. It lists the requirements for interoperability and defines two reference implementations using state diagrams. Sub clause 8.5 defines the discovery protocol used to configure the virtual machine with the information it needs to communicate with its remote peers. Sub clause 8.6 describes how the protocol can be extended for future needs. Finally, 8.7 describes how to implement DDS QoS and some advanced DDS features using RTPS.

The only purpose of introducing the RTPS virtual machine is to describe the protocol in a complete and unambiguous manner. This description is not intended to constrain the internal implementation in any way. The only criteria for a compliant implementation is that the externally-observable behavior satisfies the requirements for interoperability. In particular, an implementation could be based on other classes and could use programming constructs other than state machines to implement the RTPS protocol.

8.2 Structure Module

This sub clause describes the structure of the RTPS entities that are the communication actors. The main classes used by the RTPS protocol are shown in Figure 8.1.

8.2.1 Overview

RTPS entities are the protocol-level endpoints used by the application-visible DDS entities in order to communicate with each other.

Each RTPS Entity is in a one-to-one correspondence with a DDS Entity. The HistoryCache forms the interface between the DDS Entities and their corresponding RTPS Entities. For example, each write operation on a DDS DataWriter adds a CacheChange to the HistoryCache of its corresponding RTPS Writer. The RTPS Writer subsequently transfers the CacheChange to the HistoryCache of all matching RTPS Readers. On the receiving side, the DDS DataReader is notified by the RTPS Reader that a new CacheChange has arrived in the HistoryCache, at which point the DDS DataReader may choose to access it using the DDS read or take API.
This sub clause provides an overview of the main classes used by the RTPS virtual machine and the types used to describe their attributes. Subsequent sub clauses describe each class in detail.

### 8.2.1.1 Summary of the classes used by the RTPS virtual machine

All RTPS entities derive from the RTPS `Entity` class. Table 8.1 lists the classes used by the RTPS virtual machine.

<table>
<thead>
<tr>
<th>RTPS Entities and Classes</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Entity</strong></td>
<td>Base class for all RTPS entities. RTPS <code>Entity</code> represents the class of objects that are visible to other RTPS Entities on the network. As such, RTPS <code>Entity</code> objects have a globally-unique identifier (GUID) and can be referenced inside RTPS messages.</td>
</tr>
<tr>
<td><strong>Endpoint</strong></td>
<td>Specialization of RTPS <code>Entity</code> representing the objects that can be communication endpoints. That is, the objects that can be the sources or destinations of RTPS messages.</td>
</tr>
<tr>
<td><strong>Participant</strong></td>
<td>Container of all RTPS entities that share common properties and are located in a single address space.</td>
</tr>
</tbody>
</table>
Writer | Specialization of RTPS Endpoint representing the objects that can be the sources of messages communicating CacheChanges.

Reader | Specialization of RTPS Endpoint representing the objects that can be used to receive messages communicating CacheChanges.

HistoryCache | Container class used to temporarily store and manage sets of changes to data-objects. On the Writer side it contains the history of the changes to data-objects made by the Writer. It is not necessary that the full history of all changes ever made is maintained. Rather what is needed is the partial history required to service existing and future matched RTPS Reader endpoints. The partial history needed depends on the DDS QoS and the state of the communications with the matched Reader endpoints. On the Reader side it contains the history of the changes to data-objects made by the matched RTPS Writer endpoints. It is not necessary that the full history of all changes ever received is maintained. Rather what is needed is a partial history containing the superposition of the changes received from the matched writers as needed to satisfy the needs of the corresponding DDS DataReader. The rules for this superposition and the amount of partial history required depend on the DDS QoS and the state of the communication with the matched RTPS Writer endpoints.

CacheChange | Represents an individual change made to a data-object. Includes the creation, modification, and deletion of data-objects.

Data | Represents the data that may be associated with a change made to a data-object.

8.2.1.2 Summary of the types used to describe RTPS Entities and Classes

The Entities and Classes used by the virtual machine each contain a set of attributes. The types of the attributes are summarized in Table 8.2.

Table 8.2 - Types of the attributes that appear in the RTPS Entities and Classes

<table>
<thead>
<tr>
<th>Attribute type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUID_t</td>
<td>Type used to hold globally-unique RTPS-entity identifiers. These are identifiers used to uniquely refer to each RTPS Entity in the system. Must be possible to represent using 16 octets. The following values are reserved by the protocol: GUID_UNKNOWN.</td>
</tr>
<tr>
<td>GuidPrefix_t</td>
<td>Type used to hold the prefix of the globally-unique RTPS-entity identifiers. The GUIDs of entities belonging to the same participant all have the same prefix (see 8.2.4.4). Must be possible to represent using 12 octets. The following values are reserved by the protocol: GUIDPREFIX_UNKNOWN.</td>
</tr>
<tr>
<td>EntityId_t</td>
<td>Type used to hold the suffix part of the globally-unique RTPS-entity identifiers. The EntityId_t uniquely identifies an Entity within a Participant. Must be possible to represent using 4 octets. The following values are reserved by the protocol: ENTITYID_UNKNOWN. Additional pre-defined values are defined by the Discovery module in 8.5.</td>
</tr>
<tr>
<td>SequenceNumber_t</td>
<td>Type used to hold sequence numbers. Must be possible to represent using 64 bits. The following values are reserved by the protocol: SEQUENCENUMBER_UNKNOWN.</td>
</tr>
<tr>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Locator_t</td>
<td>Type used to represent the addressing information needed to send a message to an RTPS Endpoint using one of the supported transports. Should be able to hold a discriminator identifying the kind of transport, an address, and a port number. It must be possible to represent the discriminator and port number using 4 octets each, the address using 16 octets. The following values are reserved by the protocol: LOCATOR_INVALID LOCATOR_KIND_INVALID LOCATOR_KIND_RESERVED LOCATOR_KIND_UDPv4 LOCATOR_KIND_UDPv6 LOCATOR_ADDRESS_INVALID LOCATOR_PORT_INVALID</td>
</tr>
<tr>
<td>TopicKind_t</td>
<td>Enumeration used to distinguish whether a Topic has defined some fields within to be used as the ‘key’ that identifies data-instances within the Topic. See the DDS specification for more details on keys. The following values are reserved by the protocol: NO_KEY WITH_KEY</td>
</tr>
<tr>
<td>ChangeKind_t</td>
<td>Enumeration used to distinguish the kind of change that was made to a data-object. Includes changes to the data or the instance state of the data-object. It can take the values: ALIVE, ALIVE_FILTERED, NOT_ALIVE_DISPOSED, NOT_ALIVE_UNREGISTERED</td>
</tr>
<tr>
<td>ChangeCount_t</td>
<td>Type used to hold a counter representing the number of HistoryCache changes that belong to a certain category. For example, the number of changes that have been filtered for an RTPS Reader endpoint.</td>
</tr>
<tr>
<td>ReliabilityKind_t</td>
<td>Enumeration used to indicate the level of the reliability used for communications. It can take the values: BEST_EFFORT, RELIABLE.</td>
</tr>
<tr>
<td>InstanceHandle_t</td>
<td>Type used to represent the identity of a data-object whose changes in value are communicated by the RTPS protocol.</td>
</tr>
<tr>
<td>ProtocolVersion_t</td>
<td>Type used to represent the version of the RTPS protocol. The version is composed of a major and a minor version number. See also 8.6. The following values are reserved by the protocol: PROTOCOLVERSION PROTOCOLVERSION_1_0 PROTOCOLVERSION_1_1 PROTOCOLVERSION_2_0 PROTOCOLVERSION_2_1 PROTOCOLVERSION_2_2 PROTOCOLVERSION_2_4 PROTOCOLVERSION is an alias for the most recent version, in this case PROTOCOLVERSION_2_4</td>
</tr>
<tr>
<td>VendorId_t</td>
<td>Type used to represent the vendor of the service implementing the RTPS protocol. The possible values for the vendorId are assigned by the OMG. The following values are reserved by the protocol: VENDORID_UNKNOWN</td>
</tr>
</tbody>
</table>
8.2.1.3 Configuration attributes of the RTPS Entities

RTPS entities are configured by a set of attributes. Some of these attributes map to the QoS policies set on the corresponding DDS entities. Other attributes represent parameters that allow tuning the behavior of the protocol to specific transport and deployment situations. Additional attributes encode the state of the RTPS Entity and are not used to configure the behavior.

The attributes used to configure a subset of the RTPS Entities are shown in Figure 8.2. The attributes to configure Writer and Reader Entities are closely tied to the protocol behavior and will be introduced in 8.4.

Figure 8.2 - Attributes used to configure the main RTPS Entities

The remainder of this sub clause describes each of the RTPS entities in more detail.

8.2.2 The RTPS HistoryCache

The HistoryCache is part of the interface between DDS and RTPS and plays different roles on the reader and the writer side.

On the writer side, the HistoryCache contains the partial history of changes to data-objects made by the corresponding DDS Writer that are needed to service existing and future matched RTPS Reader endpoints. The partial history needed depends on the DDS Qos and the state of the communications with the matched RTPS Reader endpoints.

On the reader side, it contains the partial superposition of changes to data-objects made by all the matched RTPS Writer endpoints.

The word “partial” is used to indicate that it is not necessary that the full history of all changes ever made is maintained. Rather what is needed is the subset of the history needed to meet the behavioral needs of the RTPS protocol and the QoS needs of the related DDS entities. The rules that define this subset are defined by the RTPS protocol and depend both on the state of the communications protocol and on the QoS of the related DDS entities.

The HistoryCache is part of the interface between DDS and RTPS. In other words, both the RTPS entities and their related DDS entities are able to invoke the operations on their associated HistoryCache.
The `HistoryCache` attributes are listed in Table 8.3.

Table 8.3 – RTPS HistoryCache Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>meaning</th>
<th>relation to DDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>changes</td>
<td>CacheChange[*]</td>
<td>The list of CacheChanges contained in the HistoryCache.</td>
<td>N/A.</td>
</tr>
</tbody>
</table>

The RTPS entities and the related DDS entities interact with the `HistoryCache` using the operations in Table 8.4.

Table 8.4 - RTPS HistoryCache operations

<table>
<thead>
<tr>
<th>operation name</th>
<th>parameter list</th>
<th>parameter type</th>
</tr>
</thead>
<tbody>
<tr>
<td>new</td>
<td>&lt;return value&gt;</td>
<td>HistoryCache</td>
</tr>
<tr>
<td>add_change</td>
<td>&lt;return value&gt;</td>
<td>void</td>
</tr>
<tr>
<td>a_change</td>
<td>CacheChange</td>
<td></td>
</tr>
</tbody>
</table>
The following sub clauses provide details on the operations.

8.2.2.1 new

This operation creates a new RTPS HistoryCache. The newly-created history cache is initialized with an empty list of changes.

8.2.2.2 add_change

This operation inserts the CacheChange a_change into the HistoryCache.

This operation will only fail if there are not enough resources to add the change to the HistoryCache. It is the responsibility of the DDS service implementation to configure the HistoryCache in a manner consistent with the DDS Entity RESOURCE_LIMITS QoS and to propagate any errors to the DDS-user in the manner specified by the DDS specification.

This operation performs the following logical steps:

ADD a_change TO this.changes;

8.2.2.3 remove_change

This operation indicates that a previously-added CacheChange should be removed from the HistoryCache and the details regarding the CacheChange need not be maintained in the HistoryCache. The determination of which changes should be removed from the cache is made based on the QoS associated with the related DDS entity and on the acknowledgment status of the CacheChange. This is described in 8.4.1.

This operation performs the following logical steps:

REMOVE a_change FROM this.changes;

8.2.2.4 get_seq_num_min

This operation retrieves the smallest value of the CacheChange::sequenceNumber attribute among the CacheChange stored in the HistoryCache. This operation performs the following logical steps:

min_seq_num := MIN { change.sequenceNumber WHERE (change IN this.changes) }
return min_seq_num;

8.2.2.5 get_seq_num_max

This operation retrieves the largest value of the CacheChange::sequenceNumber attribute among the CacheChange stored in the HistoryCache.

This operation performs the following logical steps:

max_seq_num := MAX { change.sequenceNumber WHERE (change IN this.changes) }
return max_seq_num;

8.2.3 The RTPS CacheChange

Class used to represent each change added to the HistoryCache. The CacheChange attributes are listed in Table 8.5.

<table>
<thead>
<tr>
<th>remove_change</th>
<th>&lt;return value&gt;</th>
<th>void</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_change</td>
<td>CacheChange</td>
<td></td>
</tr>
<tr>
<td>get_seq_num_min</td>
<td>&lt;return value&gt;</td>
<td>SequenceNumber_t</td>
</tr>
<tr>
<td>get_seq_num_max</td>
<td>&lt;return value&gt;</td>
<td>SequenceNumber_t</td>
</tr>
</tbody>
</table>
### Table 8.5 - RTPS CacheChange attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>meaning</th>
<th>relation to DDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>ChangeKind_t</td>
<td>Identifies the kind of change. See Table 8.2</td>
<td>DDS instance state kind</td>
</tr>
<tr>
<td>writerGuid</td>
<td>GUID_t</td>
<td>The GUID_t that identifies the RTPS Writer that made the change</td>
<td>N/A.</td>
</tr>
<tr>
<td>instanceHandle</td>
<td>InstanceHandle_t</td>
<td>Identifies the instance of the data-object to which the change applies.</td>
<td>In DDS, the value of the fields labeled as ‘key’ within the data uniquely identify each data- object.</td>
</tr>
<tr>
<td>sequenceNumber</td>
<td>SequenceNumber_t</td>
<td>Sequence number assigned by the RTPS Writer to uniquely identify the change.</td>
<td>N/A.</td>
</tr>
<tr>
<td>data_value</td>
<td>Data</td>
<td>The data value associated with the change. Depending on the kind of CacheChange, there may be no associated data. See Table 8.2.</td>
<td>N/A.</td>
</tr>
<tr>
<td>inlineQos</td>
<td>ParameterList</td>
<td>Contains QoS that may affect the interpretation of the CacheChange::data_value.</td>
<td>DDS-specific information which affects the data.</td>
</tr>
</tbody>
</table>

### 8.2.4 The RTPS Entity

RTPS *Entity* is the base class for all RTPS entities and maps to a DDS Entity. The *Entity* configuration attributes are listed in Table 8.6.

#### Table 8.6 - RTPS Entity Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>meaning</th>
<th>relation to DDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>guid</td>
<td>GUID_t</td>
<td>Globally and uniquely identifies the RTPS Entity within the DDS domain</td>
<td>Maps to the value of the DDS BuiltinTopicKey_t used to describe the corresponding DDS Entity. Refer to the DDS specification for more details.</td>
</tr>
</tbody>
</table>

### 8.2.4.1 Identifying RTPS entities: The GUID

The GUID (Globally Unique Identifier) is an attribute of all RTPS Entities and uniquely identifies the Entity within a DDS Domain.

The GUID is built as a tuple <prefix, entityId> combining a *GuidPrefix_t* prefix and an *EntityId_t* entityId.
8.2.4.2 The GUIDs of RTPS Participants

Every Participant has GUID <prefix, ENTITYID_PARTICIPANT>, where the constant ENTITYID_PARTICIPANT is a special value defined by the RTPS protocol. Its actual value depends on the PSM.

The implementation is free to choose the prefix, as long as every Participant in the Domain has a unique GUID.

8.2.4.3 The GUIDs of Endpoint Groups within a Participant

The endpoint Groups contained by a Participant with GUID <participantPrefix, ENTITYID_PARTICIPANT> have the GUID <participantPrefix, entityId>. The entityId is the unique identification of the Group relative to the Participant. This has several consequences:

- The GUIDs of all the Groups within a Participant have the same prefix.
- Once the GUID of a Group is known, the GUID of the Participant that contains the endpoint is also known.
- The GUID of any Group can be deduced from the GUID of the Participant to which it belongs and its entityId. The selection of entityId for each RTPS Entity depends on the PSM.

8.2.4.4 The GUIDs of the RTPS Endpoints within a Participant

The Endpoints contained by a Participant with GUID <participantPrefix, ENTITYID_PARTICIPANT> have the GUID <participantPrefix, entityId>. The entityId is the unique identification of the Endpoint relative to the Participant. This has several consequences:

- The GUIDs of all the Endpoints within a Participant have the same prefix.
- Once the GUID of an Endpoint is known, the GUID of the Participant that contains the endpoint is also known.
- The GUID of any endpoint can be deduced from the GUID of the Participant to which it belongs and its entityId. The selection of entityId for each RTPS Entity depends on the PSM.

8.2.5 The RTPS Participant

RTPS Participant is the container of RTPS Group entities which contain Endpoint entities. The RTPS Participant maps to a DDS DomainParticipant.
RTPS Participant contains the attributes shown in Table 8.8.

### Table 8.8 - RTPS Participant attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>meaning</th>
<th>relation to DDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>defaultUnicastLocatorList</td>
<td>Locator_t[*]</td>
<td>Default list of unicast locators (transport, address, port combinations) that can be used to send messages to the Endpoints contained in the Participant. These are the unicast locators that will be used in case the Endpoint does not specify its own set of Locators.</td>
<td>N/A. Configured by discovery</td>
</tr>
<tr>
<td>defaultMulticastLocatorList</td>
<td>Locator_t[*]</td>
<td>Default list of multicast locators (transport, address, port combinations) that can be used to send messages to the Endpoints contained in the Participant. These are the multicast locators that will be used in case the Endpoint does not specify its own set of Locators.</td>
<td>N/A. Configured by discovery</td>
</tr>
<tr>
<td>protocolVersion</td>
<td>ProtocolVersion_t</td>
<td>Identifies the version of the RTPS protocol that the Participant uses to communicate.</td>
<td>N/A. Specified for each version of the protocol.</td>
</tr>
<tr>
<td>vendorId</td>
<td>VendorId_t</td>
<td>Identifies the vendor of the RTPS middleware that contains the Participant.</td>
<td>N/A. Configured by each vendor.</td>
</tr>
</tbody>
</table>

### 8.2.6 The RTPS Group

RTPS Group is a container for RTPS Endpoint entities. It provides a way for RTPS Endpoint entities to share common properties.

There are two kinds of RTPS Group entities: Publisher and Subscriber.
The RTPS Publisher contains RTPS Writer endpoints. The RTPS Publisher maps to a DDS Publisher.

The RTPS Subscriber contains RTPS Reader endpoints. The RTPS Subscriber maps to a DDS Subscriber.

### 8.2.7 The RTPS Endpoint

RTPS Endpoint represents the possible communication endpoints from the point of view of the RTPS protocol. There are two kinds of RTPS Endpoint entities: Writer endpoints and Reader endpoints.

RTPS Writer endpoints send CacheChange messages to RTPS Reader endpoints and potentially receive acknowledgments for the changes they send. RTPS Reader endpoints receive CacheChange and change-availability announcements from Writer endpoints and potentially acknowledge the changes and/or request missed changes.

RTPS Endpoint contains the attributes shown in Table 8.9.

Table 8.9 - RTPS Endpoint configuration attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>meaning</th>
<th>relation to DDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>unicastLocatorList</td>
<td>Locator_t[*]</td>
<td>List of unicast locators (transport, address, port combinations) that can be used to send messages to the Endpoint. The list may be empty.</td>
<td>N/A. Configured by discovery</td>
</tr>
<tr>
<td>multicastLocatorList</td>
<td>Locator_t[*]</td>
<td>List of multicast locators (transport, address, port combinations) that can be used to send messages to the Endpoint. The list may be empty.</td>
<td>N/A. Configured by discovery</td>
</tr>
<tr>
<td>reliabilityLevel</td>
<td>ReliabilityKind_t</td>
<td>The levels of reliability supported by the Endpoint.</td>
<td>Maps to the RELIABILITY QoS ‘kind.’</td>
</tr>
<tr>
<td>topicKind</td>
<td>TopicKind_t</td>
<td>Used to indicate whether the Endpoint supports instance lifecycle management operations (see 8.7.4).</td>
<td>Defined by the Data-type that is associated with the DDS Topic related to the RTPS Endpoint. Indicates whether the Endpoint is associated with a DataType that has defined some fields as containing the DDS key.</td>
</tr>
<tr>
<td>endpointGroup</td>
<td>EntityId_t</td>
<td>Used to identify the RTPS Group (Publisher or Subscriber) to which the Endpoint belongs.</td>
<td>Identifies the DDS Publisher or Subscriber associated with the RTPS Endpoint.</td>
</tr>
</tbody>
</table>

### 8.2.8 The RTPS Writer

RTPS Writer specializes RTPS Endpoint and represents the actor that sends CacheChange messages to the matched RTPS Reader endpoints. Its role is to transfer all CacheChange changes in its HistoryCache to the HistoryCache of the matching remote RTPS Readers.
An RTPS **Writer** belongs to an RTPS **Group** of type **Publisher**.

The attributes to configure an RTPS **Writer** are closely tied to the protocol behavior and will be introduced in the Behavior Module (8.4).

### 8.2.9 The RTPS Reader

RTPS **Reader** specializes RTPS **Endpoint** and represents the actor that receives **CacheChange** messages from the matched RTPS **Writer** endpoints.

An RTPS **Reader** belongs to an RTPS **Group** of type **Subscriber**.

The attributes to configure an RTPS **Reader** are closely tied to the protocol behavior and will be introduced in the Behavior Module (8.4).

### 8.2.10 Relation to DDS Entities

As mentioned in 8.2.2, the **HistoryCache** forms the interface between DDS Entities and their corresponding RTPS Entities. A DDS DataWriter, for example, passes data to its matching RTPS **Writer** through the common **HistoryCache**.

How exactly a DDS Entity interacts with the **HistoryCache** however, is implementation specific and not formally modeled by the RTPS protocol. Instead, the Behavior Module of the RTPS protocol *only* specifies how **CacheChange** changes are transferred from the **HistoryCache** of the RTPS **Writer** to the **HistoryCache** of each matching RTPS **Reader**.

Despite the fact that it is not part of the RTPS protocol, it is important to know how a DDS Entity may interact with the **HistoryCache** to obtain a complete understanding of the protocol. This topic forms the subject of this sub clause.

The interactions are described using UML state diagrams. The abbreviations used to refer to DDS and RTPS Entities are listed in Table 8.10 below.

**Table 8.10 - Abbreviations used in the sequence charts and state diagrams**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Meaning</th>
<th>Example usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>DW</td>
<td>DDS DataWriter</td>
<td>DW::write</td>
</tr>
<tr>
<td>DR</td>
<td>DDS DataReader</td>
<td>DR::read</td>
</tr>
<tr>
<td>W</td>
<td>RTPS Writer</td>
<td>W::heartbeatPeriod</td>
</tr>
<tr>
<td>R</td>
<td>RTPS Reader</td>
<td>R::heartbeatResponseDelay</td>
</tr>
<tr>
<td>WHC</td>
<td>HistoryCache of RTPS Writer</td>
<td>WHC::changes</td>
</tr>
<tr>
<td>RHC</td>
<td>HistoryCache of RTPS Reader</td>
<td>RHC::changes</td>
</tr>
</tbody>
</table>

### 8.2.10.1 The DDS DataWriter

The write operation on a DDS DataWriter adds **CacheChange** changes to the **HistoryCache** of its associated RTPS **Writer**. As such, the **HistoryCache** contains a history of the most recently written changes. The number of changes is determined by QoS settings on the DDS DataWriter such as the HISTORY and RESOURCE_LIMITS QoS.

By default, all changes in the **HistoryCache** are considered relevant for each matching remote RTPS **Reader**. That is, the **Writer** should attempt to send all changes in the **HistoryCache** to the matching remote **Readers**. How to do this is the subject of the Behavior Module of the RTPS protocol.
Changes may not be sent to a remote Reader for two reasons:

- they have been removed from the HistoryCache by the DDS DataWriter and are no longer available.
- they are considered not relevant to this Reader.

The DDS DataWriter may decide to remove changes from the HistoryCache for several reasons. For example, only a limited number of changes may need to be stored based on the HISTORY QoS settings. Alternatively, a sample may have expired due to the LIFESPAN QoS. When using strict reliable communication, a change can only be removed when it has been acknowledged by all readers the change was sent to and which are still active and alive.

Not all changes may be relevant for each matching remote Reader as determined by, for example, the TIME_BASED_FILTER QoS or through the use of DDS content-filtered topics. Note that whether a change is relevant must be determined on a per Reader basis in this case. Implementations may be able to optimize bandwidth and/or CPU usage by filtering on the Writer side when possible. Whether this is possible depends on whether an implementation keeps track of each individual remote Reader and the QoS and filters that apply to this Reader. The Reader itself will always filter.

QoS or content-based filtering is represented in this document using DDS_FILTER(reader, change), a notation which reflects that filtering is reader dependent. Depending on what reader specific information is stored by the writer, DDS_FILTER may be a noop. This operation returns 'true' if the change passes the filter and should be sent to the reader. Otherwise it returns 'false'. For content-based filtering, the RTPS specification enables sending information with each change that lists what filters have been applied to the change and which filters it passed. If available, this information can then be used by the Reader to filter a change without having to call DDS_FILTER. This approach saves CPU cycles by filtering the sample once on the Writer side, as opposed to filtering on each Reader.

The following state-diagram illustrates how the DDS Data Writer adds a change to the HistoryCache.

Figure 8.6 - DDS DataWriter additions to the HistoryCache
Table 8.11 - Transitions for DDS DataWriter additions to the HistoryCache

<table>
<thead>
<tr>
<th>Transition</th>
<th>state</th>
<th>event</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>initial</td>
<td>new DDS DataWriter</td>
<td>alive</td>
</tr>
<tr>
<td>T2</td>
<td>alive</td>
<td>DataWriter::write</td>
<td>alive</td>
</tr>
<tr>
<td>T3</td>
<td>alive</td>
<td>DataWriter::dispose</td>
<td>alive</td>
</tr>
<tr>
<td>T4</td>
<td>alive</td>
<td>DataWriter::unregister</td>
<td>alive</td>
</tr>
<tr>
<td>T5</td>
<td>alive</td>
<td>delete DDS DataWriter</td>
<td>final</td>
</tr>
</tbody>
</table>

8.2.10.1.1 Transition T1

This transition is triggered by the creation of a DDS DataWriter ‘the_dds_writer.’ The transition performs the following logical actions in the virtual machine:

```plaintext
the_rtps_writer = new RTPS::Writer;
the_dds_writer.related_rtps_writer := the_rtps_writer;
```

8.2.10.1.2 Transition T2

This transition is triggered by the act of writing data using a DDS DataWriter ‘the_dds_writer.’ The DataWriter::write() operation takes as arguments the ‘data’ and the InstanceHandle_t ‘handle’ used to differentiate among different data-objects.

The transition performs the following logical actions in the virtual machine:

```plaintext
the_rtps_writer := the_dds_writer.related_rtps_writer;
a_change := the_rtps_writer.new_change(ALIVE, data, inlineQos, handle);
the_rtps_writer.writer_cache.add_change(a_change);
```

After the transition the following post-conditions hold:

```plaintext
the_rtps_writer.writer_cache.get_seq_num_max() == a_change.sequenceNumber
```

8.2.10.1.3 Transition T3

This transition is triggered by the act of disposing a data-object previously written with the DDS DataWriter ‘the_dds_writer.’ The DataWriter::dispose() operation takes as parameter the InstanceHandle_t ‘handle’ used to differentiate among different data-objects.

This operation has no effect if the topicKind==NO_KEY.

The transition performs the following logical actions in the virtual machine:

```plaintext
if (the_rtps_writer.topicKind == WITH_KEY) {
    a_change := the_rtps_writer.new_change(NOT_ALIVE_DISPOSED, <nil>, inlineQos, handle);
    the_rtps_writer.writer_cache.add_change(a_change);
}
```

After the transition the following post-conditions hold:

```plaintext
if (the_rtps_writer.topicKind == WITH_KEY) then
    the_rtps_writer.writer_cache.get_seq_num_max() == a_change.sequenceNumber
```

8.2.10.1.4 Transition T4

This transition is triggered by the act of unregistering a data-object previously written with the DDS DataWriter ‘the_dds_writer.’ The DataWriter::unregister() operation takes as arguments the InstanceHandle_t ‘handle’ used to differentiate among different data-objects.

This operation has no effect if the topicKind==NO_KEY.
The transition performs the following logical actions in the virtual machine:

```c
the_rtps_writer := the.dds_writer.related_rtps_writer;
if (the_rtps_writer.topicKind == WITH_KEY) {
    a_change := the_rtps_writer.new_change(NOT_ALIVE_UNREGISTERED, <nil>,
                                             inlineQos, handle);
    the_rtps_writer.writer_cache.add_change(a_change);
}
```

After the transition the following post-conditions hold:

```c
if (the_rtps_writer.topicKind == WITH_KEY) then
    the_rtps_writer.writer_cache.get_seq_num_max() == a_change.sequenceNumber
```

### 8.2.10.1.5 Transition T5

This transition is triggered by the destruction of a DDS DataWriter ‘the.dds_writer.’ The transition performs the following logical actions in the virtual machine:

```c
delete the.dds_writer.related_rtps_writer;
```

### 8.2.10.2 The DDS DataReader

The DDS DataReader gets its data from the HistoryCache of the corresponding RTPS Reader. The number of changes stored in the HistoryCache is determined by QoS settings such as the HISTORY and RESOURCE_LIMITS QoS.

Each matching Writer will attempt to transfer all relevant samples from its HistoryCache to the HistoryCache of the Reader. The implementation of the read or take call on the DDS DataReader accesses the HistoryCache. The changes returned to the user are those in the HistoryCache that pass all Reader specific filters, if any.

A Reader filter is equally represented by DDS_FILTER(reader, change). As mentioned above, implementations may be able to perform most of the filtering on the Writer side. In that case, samples are either never sent (and therefore not present in the HistoryCache of the Reader) or contain information on what filters where applied and the corresponding outcome (for content-based filtering).

A DDS DataReader may also decide to remove changes from the HistoryCache in order to satisfy such QoS as TIME_BASED_FILTER. This exact behavior is again implementation specific and is not modeled by the RTPS protocol.

The following state-diagram illustrates how the DDS Data Reader accesses changes in the HistoryCache.

![State-Diagram DDS DataReader access to the HistoryCache](image)

Figure 8.7 - DDS DataReader access to the HistoryCache
Table 8.12 - Transitions for DDS DataReader access to the HistoryCache

<table>
<thead>
<tr>
<th>Transition</th>
<th>state</th>
<th>event</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>initial</td>
<td>new DDS DataReader</td>
<td>alive</td>
</tr>
<tr>
<td>T2</td>
<td>alive</td>
<td>DDS DataReader::read</td>
<td>alive</td>
</tr>
<tr>
<td>T3</td>
<td>alive</td>
<td>DDS DataReader::take</td>
<td>alive</td>
</tr>
<tr>
<td>T4</td>
<td>alive</td>
<td>delete DDS DataReader</td>
<td>final</td>
</tr>
</tbody>
</table>

### 8.2.10.2.1 Transition T1

This transition is triggered by the creation of a DDS DataReader ‘the.dds_reader.’ The transition performs the following logical actions in the virtual machine:

```c
the_rtps_reader = new RTPS::Reader;
the.dds_reader.related_rtps_reader := the_rtps_reader;
```

### 8.2.10.2.2 Transition T2

This transition is triggered by the act of reading data from the DDS DataReader ‘the.dds_reader’ by means of the ‘read’ operation. Changes returned to the application remain in the RTPS Reader’s HistoryCache such that subsequent read or take operations can find them again.

The transition performs the following logical actions in the virtual machine:

```c
the_rtps_reader := the.dds_reader.related_rtps_reader;
a_change_list := new();
FOREACH change IN the_rtps_reader.reader_cache.changes {
    if DDS_FILTER(the_rtps_reader, change) {
        ADD change TO a_change_list;
    }
}
RETURN a_change_list;
```

The DDS_FILTER() operation reflects the capabilities of the DDS DataReader API to select a subset of changes based on CacheChange::kind, QoS, content-filters and other mechanisms. Note that the logical actions above only reflect the behavior and not necessarily the actual implementation of the protocol.

### 8.2.10.2.3 Transition T3

This transition is triggered by the act of reading data from the DDS DataReader ‘the.dds_reader’ by means of the ‘take’ operation. Changes returned to the application are removed from the RTPS Reader’s HistoryCache such that subsequent read or take operations do not find the same change.

The transition performs the following logical actions in the virtual machine:

```c
the_rtps_reader := the.dds_reader.related_rtps_reader;
a_change_list := new();
FOREACH change IN the_rtps_reader.reader_cache.changes {
    if DDS_FILTER(the_rtps_reader, change) {
        ADD change TO a_change_list;
    }
}
the_rtps_reader.reader_cache.remove_change(a_change);
RETURN a_change_list;
```

The DDS_FILTER() operation reflects the capabilities of the DDS DataReader API to select a subset of changes based on CacheChange::kind, QoS, content-filters and other mechanisms. Note that the logical actions above only reflect the behavior and not necessarily the actual implementation of the protocol.

After the transition the following post-conditions hold:

```c
FOREACH change IN a_change_list
```
8.2.10.2.4 Transition T4

This transition is triggered by the destruction of a DDS DataReader ‘the_dds_reader.’ The transition performs the following logical actions in the virtual machine:

```c
delete the_dds_reader.related_rtps_reader;
```

8.3 Messages Module

The Messages module describes the overall structure and logical contents of the messages that are exchanged between the RTPS Writer endpoints and RTPS Reader endpoints. RTPS Messages are modular by design and can be easily extended to support both standard protocol feature additions as well as vendor-specific extensions.

8.3.1 Overview

The Messages module is organized as follows:

- 8.3.2 introduces any additional types needed for defining RTPS messages in the subsequent sub clauses.
- 8.3.3 describes the common structure used for all RTPS Messages. All RTPS Messages consist of a Header followed by a series of Submessages. The number of Submessages that can be sent in a single RTPS Message is only limited by the maximum message size the underlying transport can support.
- Certain Submessages may affect how subsequent Submessages within the same RTPS Message must be interpreted. The context for interpreting Submessages is maintained by the RTPS Message Receiver and is described in 8.3.4.
- 8.3.5 lists the elementary building blocks for creating Submessages, also referred to as SubmessageElements. This includes sequence number sets, timestamp, identifiers, etc.
- 8.3.6 describes the structure of the RTPS Header. The fixed size RTPS Header is used to identify an RTPS Message.
- Finally, 8.3.8 introduces all available Submessages in detail. For each Submessage, the specification defines its contents, when it is considered valid and how it affects the state of the RTPS Message Receiver. The PSM will define the actual mapping of each of these Submessage to bits and bytes on the wire in 9.4.5.

8.3.2 Type Definitions

In addition to the types defined in 8.2.1.2, the Messages module makes use of the types listed in Table 8.13.

**Table 8.13 - Types used to define RTPS messages**

<table>
<thead>
<tr>
<th>Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProtocolId_t</td>
<td>Enumeration used to identify the protocol. The following values are reserved by the protocol: PROTOCOL_RTPS</td>
</tr>
<tr>
<td>SubmessageFlag</td>
<td>Type used to specify a Submessage flag. A Submessage flag takes a boolean value and affects the parsing of the Submessage by the receiver.</td>
</tr>
<tr>
<td>SubmessageKind</td>
<td>Enumeration used to identify the kind of Submessage. The following values are reserved by this version of the protocol: RTPS_HE, DATA, GAP, HEARTBEAT, ACKNACK, PAD, INFO_TS, INFO_REPLY, INFO_DST, INFO_SRC, DATA_FRAG, NACK_FRAG, HEARTBEAT_FRAG</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Time_t</td>
<td>Type used to hold a timestamp. Should have at least nano-second resolution. The following values are reserved by the protocol: TIME_ZERO, TIME_INVALID, TIME_INFINITE</td>
</tr>
<tr>
<td>Count_t</td>
<td>Type used to hold a count that is incremented monotonically, used to identify message duplicates.</td>
</tr>
<tr>
<td>Checksum_t</td>
<td>Type used to hold a checksum. Used to detect RTPS message corruption by the underlying transport. The following values are reserved by the protocol: CHECKSUM_INVALID.</td>
</tr>
<tr>
<td>MessageLength_t</td>
<td>Type used to hold the length of an RTPS Message. The following values are reserved by the protocol: MESSAGE_LENGTH_INVALID</td>
</tr>
<tr>
<td>ParameterId_t</td>
<td>Type used to uniquely identify a parameter in a parameter list. Used extensively by the Discovery Module mainly to define QoS Parameters. A range of values is reserved for protocol-defined parameters, while another range can be used for vendor-defined parameters, see 8.3.5.9.</td>
</tr>
<tr>
<td>FragmentNumber_t</td>
<td>Type used to hold fragment numbers. Must be possible to represent using 32 bits.</td>
</tr>
<tr>
<td>GroupDigest_t</td>
<td>Type used to hold a digest value that uniquely identifies a group of Entities belonging to the same Participant.</td>
</tr>
<tr>
<td>UExtension4_t</td>
<td>Type used to hold an undefined 4-byte value. It is intended to be used in future revisions of the specification.</td>
</tr>
<tr>
<td>WExtension8_t</td>
<td>Type used to hold an undefined 8-byte value. It is intended to be used in future revisions of the specification.</td>
</tr>
</tbody>
</table>

### 8.3.3 The Overall Structure of an RTPS Message

The overall structure of an RTPS Message consists of a fixed-size leading RTPS Header followed by a variable number of RTPS Submessage parts. Each Submessage in turn consists of a SubmessageHeader and a variable number of SubmessageElements. This is illustrated in Figure 8.8.
Each message sent by the RTPS protocol has a finite length. This length is optionally sent in the RTPS **HeaderExtension** Submessage.

The length may also be sent by the underlying transport that carries the RTPS message. In this case it may be omitted from the **HeaderExtension**. For example, in the case of a packet-oriented transport (like UDP/IP), the length of the message is already provided by the transport headers. In contrast, a stream-oriented transport (like TCP) would need to include the length in the RTPS **HeaderExtension** in order to identify the boundary of the RTPS message.

### 8.3.3.1 Header structure

The RTPS **Header** must appear at the beginning of every message.
The **Header** identifies the message as belonging to the RTPS protocol. The **Header** identifies the version of the protocol and the vendor that sent the message. The **Header** contains the fields listed in Table 8.14.

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>protocol</td>
<td>ProtocolId_t</td>
<td>Identifies the message as an RTPS message.</td>
</tr>
<tr>
<td>version</td>
<td>ProtocolVersion_t</td>
<td>Identifies the version of the RTPS protocol.</td>
</tr>
<tr>
<td>vendorId</td>
<td>VendorId_t</td>
<td>Indicates the vendor that provides the implementation of the RTPS protocol.</td>
</tr>
<tr>
<td>guidPrefix</td>
<td>GuidPrefix_t</td>
<td>Defines a default prefix to use for all GUIDs that appear in the message.</td>
</tr>
</tbody>
</table>

The structure of the RTPS **Header** cannot be changed in this major version (2) of the protocol.

### 8.3.3.1.1 protocol

The **protocol** identifies the message as an RTPS message. This value is set to PROTOCOL_RTPS.

### 8.3.3.1.2 version

The **version** identifies the version of the RTPS protocol. Implementations following this version of the document implement protocol version 2.5 (major = 2, minor = 5) and have this field set to PROTOCOLVERSION.

### 8.3.3.1.3 vendorId

The **vendorId** identifies the vendor of the middleware that implemented the RTPS protocol and allows this vendor to add specific extensions to the protocol. The **vendorId** does not refer to the vendor of the device or product that contains RTPS middleware. The possible values for the **vendorId** are assigned by the OMG.

The protocol reserves the following value:

`VENDORID_UNKNOWN`

Vendor IDs can only be reserved by implementers that commit to comply with the current major version of the protocol. To facilitate incremental evolution, the list of vendor IDs is managed separately from this specification. The list is maintained on the OMG DDS website and is accessible at:


Requests for new vendor IDs should be sent via email to dds@omg.org

### 8.3.3.1.4 guidPrefix

The **guidPrefix** defines a default prefix that can be used to reconstruct the Globally Unique Identifiers (GUIDs) that appear within the Submessages contained in the message. The **guidPrefix** allows Submessages to contain only the EntityId part of the GUID and therefore saves from having to repeat the common prefix on every GUID (See 8.2.4.1).

### 8.3.3.2 HeaderExtension structure

The **HeaderExtension** may optionally appear immediately following the **Header**. It extends the information provided in the **Header** without breaking interoperability with earlier versions of the RTPS protocol.
The **HeaderExtension** submessage was introduced in RTPS version 2.5. Earlier versions of the protocol (RTPS 2.4 and earlier) do not understand the **HeaderExtension** submessage. However, since the **HeaderExtension** conforms to the sub-message structure (see 8.3.3.3) versions of the protocol that do not understand the **HeaderExtension** will treat it as “unknown message kind”, skip it, and continue processing the submessages that follow it, see 8.3.4.1.

Table 8.15 - Structure of the HeaderExtension Submessage

<table>
<thead>
<tr>
<th>element</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EndiannessFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates endianness.</td>
</tr>
<tr>
<td>LengthFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates the messageLength field is present.</td>
</tr>
<tr>
<td>TimestampFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates the rtpsSendTimestamp field is present.</td>
</tr>
<tr>
<td>UExtensionFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates the uExtension4 field is present.</td>
</tr>
<tr>
<td>WExtensionFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates the wExtension8 field is present.</td>
</tr>
<tr>
<td>ChecksumFlags</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates the presence and format of the messageChecksum field.</td>
</tr>
<tr>
<td>ParametersFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates the parameters field is present.</td>
</tr>
<tr>
<td>messageLength</td>
<td>MessageLength</td>
<td>Present only if the LengthFlag is set in the header. Contains the length of the RTPS Message.</td>
</tr>
</tbody>
</table>
### 8.3.3.3.1 messageLength

The `messageLength` indicates the length of the complete RTPS Message, starting from the beginning of the RTPS Header.

### 8.3.3.3.2 rtpsSendTimestamp

The `rtpsSendTimestamp` indicates the time when the RTPS message was sent by the originating Participant.

The timestamp may be used by the receiving application to estimate the time offset between the clocks of the sending and receiving Participants. For this reason, the `rtpsSendTimestamp` shall collected as close as possible to the moment when the RTPS Message is sent over the underlying transport.

The time origin used for the `HeaderExtension rtpsSendTimestamp` shall be the same as the one used in the Timestamp Submessage Element (see 8.3.5.8).

### 8.3.3.3.3 uExtension4

The `uExtension4` is undefined in this version of the protocol. It is intended for future revisions.

### 8.3.3.3.4 wExtension8

The `wExtension8` is undefined in this version of the protocol. It is intended for future revisions.

### 8.3.3.3.5 messageChecksum

The `messageChecksum` provides a checksum computed over the complete of the RTPS Message, which includes the RTPS Header, the `HeaderExtension`, and all Submessages that may follow.

### 8.3.3.3.6 parameters

The `parameters` provide an extensibility mechanism. It enables revisions of the protocol to add information to the `HeaderExtension` without breaking interoperability with this version, or prior versions, of the RTPS protocol.

### 8.3.3.3 Submessage structure

Each RTPS Message consists of a variable number of RTPS Submessage parts. All RTPS Submessages feature the same identical structure shown in Figure 8.11.
All Submessages start with a **SubmessageHeader** part followed by a concatenation of **SubmessageElement** parts. The **SubmessageHeader** identifies the kind of Submessage and the optional elements within that Submessage. The **SubmessageHeader** contains the fields listed in Table 8.16.

**Figure 8.11 - Structure of the RTPS Submessages**

---

### Table 8.16 - Structure of the SubmessageHeader

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>submessageId</td>
<td>SubmessageKind</td>
<td>Identifies the kind of Submessage. The possible Submessages are described in 8.3.8.</td>
</tr>
<tr>
<td>flags</td>
<td>SubmessageFlag[8]</td>
<td>Identifies the endianness used to encode the Submessage, the presence of optional elements within the Submessage, and possibly modifies the interpretation of the Submessage. There are 8 possible flags. The first flag (index 0) identifies the endianness used to encode the Submessage. The remaining flags are interpreted differently depending on the kind of Submessage and are described separately for each Submessage.</td>
</tr>
<tr>
<td>submessageLength</td>
<td>ushort</td>
<td>Indicates the length of the Submessage. Given an RTPS Message consists of a concatenation of Submessages, the Submessage length can be used to skip to the next Submessage.</td>
</tr>
</tbody>
</table>

The structure of the RTPS **Submessage** cannot be changed in this major version (2) of the protocol.

#### 8.3.3.1 SubmessageId

The **submessageId** identifies the kind of **Submessage**. The valid ID’s are enumerated by the possible values of SubmessageKind (see Table 8.13).

The meaning of the Submessage IDs cannot be modified in this major version (2). Additional Submessages can be added in higher minor versions. In order to maintain inter-operability with future versions, Platform Specific Mappings should reserve a range of values intended for protocol extensions and a range of values that are reserved for vendor-specific Submessages that will never be used by future versions of the RTPS protocol.
8.3.3.2 flags

The flags in the Submessage header contain 8 boolean values. The first flag, the EndiannessFlag, is present and located in the same position in all Submessages and represents the endianness used to encode the information in the Submessage. The literal ‘E’ is often used to refer to the EndiannessFlag.

If the EndiannessFlag is set to FALSE, the Submessage is encoded in big-endian format, EndiannessFlag set to TRUE means little-endian.

Other flags have interpretations that depend on the type of Submessage.

8.3.3.3 submessageLength

Indicates the length of the Submessage (not including the Submessage header). In case submessageLength > 0, it is either:

- The length from the start of the contents of the Submessage until the start of the header of the next Submessage (in case the Submessage is not the last Submessage in the Message).
- Or else it is the remaining Message length (in case the Submessage is the last Submessage in the Message). An interpreter of the Message can distinguish between these two cases as it knows the total length of the Message.

In case submessageLength==0, the Submessage is the last Submessage in the Message and extends up to the end of the Message. This makes it possible to send Submessages larger than 64k (the maximum length that can be stored in the submessageLength field), provided they are the last Submessage in the Message.

8.3.4 The RTPS Message Receiver

The interpretation and meaning of a Submessage within a Message may depend on the previous Submessages contained within that same Message. Therefore, the receiver of a Message must maintain state from previously deserialized Submessages in the same Message. This state is modeled as the state of an RTPS Receiver that is reset each time a new message is processed and provides context for the interpretation of each Submessage. The RTPS Receiver is shown in Figure 8.12. Table 8.17 lists the attributes used to represent the state of the RTPS Receiver.

![Figure 8.12 - RTPS Receiver](image)

For each new Message, the state of the Receiver is reset and initialized as listed below.
Table 8.17 - Initial State of the Receiver

<table>
<thead>
<tr>
<th>name</th>
<th>initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sourceVersion</td>
<td>PROTOCOLVERSION</td>
</tr>
<tr>
<td>sourceVendorId</td>
<td>VENDORID UNKNOWN</td>
</tr>
<tr>
<td>sourceGuidPrefix</td>
<td>GUIDPREFIX UNKNOWN</td>
</tr>
<tr>
<td>destGuidPrefix</td>
<td>GUID prefix of the participant receiving the message</td>
</tr>
</tbody>
</table>
| UnicastReplyLocatorList  | The list is initialized to contain a single Locator_t with the LocatorKind, Address, and Port fields specified below:  
                            • The LocatorKind is set to the kind that identifies the transport that received the message (e.g., LOCATOR_KIND_UDPv4).  
                            • The Address is set to the Address of the source of the message, assuming the Transport used supports this (e.g., for UDP the source address is part of the UDP header). Otherwise it is set to LOCATOR_ADDRESS_INVALID.  
                            • The port is set to LOCATOR_PORT_INVALID. |
| multicastReplyLocatorList| The list is initialized to contain a single Locator_t with the LocatorKind, an Address and Port fields specified below:  
                            • The LocatorKind is set to the kind that identifies the transport that received the message (e.g., LOCATOR_KIND_UDPv4).  
                            • The address is set to LOCATOR_ADDRESS_INVALID.  
                            • The port is set to LOCATOR_PORT_INVALID. |
| haveTimestamp            | FALSE                                              |
| timestamp                | TIME_INVALID                                      |
| messageLength            | MESSAGE_LENGTH_INVALID                             |
| messageChecksum          | CHECKSUM_INVALID                                   |
| rtspSendTimestamp        | TIME_INVALID                                      |
| rtspReceptionTimestamp   | TIME_INVALID                                      |
| clockSkewDetected        | FALSE                                              |
| parameters               | The list is initialized as an empty list.          |

8.3.4.1 Rules Followed by the Message Receiver

The following algorithm outlines the rules that a receiver of any Message must follow:

1. If the full Submessage header cannot be read, the rest of the Message is considered invalid.
2. The submessageLength field defines where the next Submessage starts or indicates that the Submessage extends to the end of the Message, as explained in Section 8.3.3.3.3. If this field is invalid, the rest of the Message is invalid.
3. A Submessage with an unknown SubmessageId must be ignored and parsing must continue with the next Submessage. Concretely: an implementation of RTPS 2.5 must ignore any Submessages with IDs that are outside of the SubmessageKind set defined in version 2.5. SubmessageIds in the vendor-specific range coming from a vendorId that is unknown must also be ignored and parsing must continue with the next Submessage.
4. **Submessage** flags. The receiver of a Submessage should ignore unknown flags. An implementation of RTPS 2.5 should skip all flags that are marked as “X” (unused) in the protocol.

5. A valid `submessageLength` field must **always** be used to find the next **Submessage**, even for **Submessages** with known IDs.

6. A known but invalid **Submessage** invalidates the rest of the **Message**. Sub clause 8.3.8 describes each known.

**Submessage** and when it should be considered invalid. Reception of a valid header and/or Submessage has two effects:

1. It can change the state of the Receiver; this state influences how the following **Submessages** in the **Message** are interpreted. 8.3.8 discusses how the state changes for each **Submessage**. In this version of the protocol, only the Header and the **Submessages** HeaderExtension, InfoSource, InfoReply, InfoDestination, and InfoTimestamp change the state of the Receiver.

2. It can affect the behavior of the Endpoint to which the message is destined. This applies to the basic RTPS messages: Data, DataFrag, HeartBeat, AckNack, Gap, HeartbeatFrag, NackFrag.

Sub clause 8.3.8 describes the detailed interpretation of the **Header** and every **Submessage**.

### 8.3.5 RTPS SubmessageElements

Each RTPS message contains a variable number of RTPS Submessages. Each RTPS Submessage in turn is built from a set of predefined atomic building blocks called **SubmessageElements**. RTPS 2.5 defines the submessage elements shown in Figure 8.13 below.

![Figure 8.13 - RTPS SubmessageElements](image)

#### 8.3.5.1 The GuidPrefix, and EntityId

These SubmessageElements are used to contain the **GuidPrefix_t** and **EntityId_t** parts of a GUID_t (defined in 8.2.4.1) within Submessages.

Table 8.18 - Structure of the GuidPrefix SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>GuidPrefix_t</td>
<td>Identifies the GuidPrefix_t part of the GUID_t of the Entity that is the source or target of the message.</td>
</tr>
</tbody>
</table>
Table 8.19 – Structure of the EntityId SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>EntityId_t</td>
<td>Identifies the EntityId_t part of the GUID_t of the Entity that is the source or target of the message.</td>
</tr>
</tbody>
</table>

8.3.5.2 VendorId

The VendorId identifies the vendor of the middleware implementing the RTPS protocol and allows this vendor to add specific extensions to the protocol. The vendor ID does not refer to the vendor of the device or product that contains DDS middleware.

Table 8.20 – Structure of the VendorId SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>VendorId_t</td>
<td>Identifies the vendor of the middleware that implements the protocol.</td>
</tr>
</tbody>
</table>

The following values are reserved by the protocol:

VENDORID_UNKNOWN

Other values must be assigned by the OMG.

8.3.5.3 ProtocolVersion

The ProtocolVersion defines the version of the RTPS protocol.

Table 8.21 - Structure of the ProtocolVersion SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>ProtocolVersion_t</td>
<td>Identifies the major and minor version of the RTPS protocol.</td>
</tr>
</tbody>
</table>

The RTPS protocol version 2.5 defines the following special values:

PROTOCOLVERSION_1_0
PROTOCOLVERSION_1_1
PROTOCOLVERSION_2_0
PROTOCOLVERSION_2_1
PROTOCOLVERSION_2_2
PROTOCOLVERSION_2_3
PROTOCOLVERSION_2_4

8.3.5.4 SequenceNumber

A sequence number is a 64-bit signed integer, that can take values in the range: -2^63 <= N <= 2^63-1. The selection of 64 bits as the representation of a sequence number ensures the sequence numbers never\(^1\) wrap. Sequence numbers begin at 1.

Table 8.22 – Structure of the SequenceNumber SubmessageElements

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>SequenceNumber_t</td>
<td>Provides the value of the 64-bit sequence number.</td>
</tr>
</tbody>
</table>

---

\(^1\) Even assuming an extremely fast rate of message generation for a single RTPS Writer such as 100 messages per microsecond, the 64-bit integer would not roll over for approximately 3000 years of uninterrupted operation.
The protocol reserves the following value:

SEQUENCENUMBER_UNKNOWN

8.3.5.5 SequenceNumberSet

SequenceNumberSet SubmessageElements are used as parts of several messages to provide binary information about individual sequence numbers within a range. The sequence numbers represented in the SequenceNumberSet are limited to belong to an interval with a range no bigger than 256. In other words, a valid SequenceNumberSet must verify that:

maximum(SequenceNumberSet) - minimum(SequenceNumberSet) < 256
minimum(SequenceNumberSet) >= 1

The above restriction allows SequenceNumberSet to be represented in an efficient and compact way using bitmaps.

SequenceNumberSet SubmessageElements are used for example to selectively request re-sending of a set of sequence numbers.

Table 8.23 – Structure of the SequenceNumberSet SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>SequenceNumber_t</td>
<td>Identifies the first sequence number in the set.</td>
</tr>
<tr>
<td>set</td>
<td>SequenceNumber_t[*]</td>
<td>A set of sequence numbers, each verifying that: base &lt;= element(set) &lt;= base+255</td>
</tr>
</tbody>
</table>

8.3.5.6 FragmentNumber

A fragment number is a 32-bit unsigned integer and is used by Submessages to identify a particular fragment in fragmented serialized data. Fragment numbers start at 1.

Table 8.24 - Structure of the FragmentNumber SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>FragmentNumber_t</td>
<td>Provides the value of the 32-bit fragment number.</td>
</tr>
</tbody>
</table>

8.3.5.7 FragmentNumberSet

FragmentNumberSet SubmessageElements are used to provide binary information about individual fragment numbers within a range. The fragment numbers represented in the FragmentNumberSet are limited to belong to an interval with a range no bigger than 256. In other words, a valid FragmentNumberSet must verify that:

maximum(FragmentNumberSet) - minimum(FragmentNumberSet) < 256
minimum(FragmentNumberSet) >= 1

The above restriction allows FragmentNumberSet to be represented in an efficient and compact way using bitmaps.

FragmentNumberSet SubmessageElements are used for example to selectively request re-sending of a set of fragments.

Table 8.25 - Structure of the FragmentNumberSet SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>FragmentNumber_t</td>
<td>Identifies the first fragment number in the set.</td>
</tr>
<tr>
<td>set</td>
<td>FragmentNumber_t[*]</td>
<td>A set of fragment numbers, each verifying that: base &lt;= element(set) &lt;= base+255</td>
</tr>
</tbody>
</table>
8.3.5.8 Timestamp

Timestamp is used to represent time. The representation should be capable of having a resolution of nanoseconds or better.

Table 8.26 - Structure of the Timestamp SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Time_t</td>
<td>Provides the value of the timestamp</td>
</tr>
</tbody>
</table>

There are three special values used by the protocol:

- TIME_ZERO
- TIME_INVALID
- TIME_INFINITE

8.3.5.9 ParameterList

ParameterList is used as part of several messages to contain parameters that may affect the interpretation of the message. The representation of the parameters follows a mechanism that allows extensions without breaking backwards compatibility.

Table 8.27 - Structure of the ParameterList SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter</td>
<td>Parameter[*]</td>
<td>List of parameters</td>
</tr>
</tbody>
</table>

Table 8.28 - Structure of each Parameter in a ParameterList SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameterId</td>
<td>ParameterId_t</td>
<td>Uniquely identifies a parameter</td>
</tr>
<tr>
<td>length</td>
<td>short</td>
<td>Length of the parameter value</td>
</tr>
<tr>
<td>value</td>
<td>octet[length]</td>
<td>Parameter value</td>
</tr>
</tbody>
</table>

The actual representation of the ParameterList is defined for each PSM. However, in order to support interoperability or bridging between PSMs and allow for extensions that preserve backwards compatibility, the representation used by all PSMs must comply with the following rules:

- There shall be no more than $2^{16}$ possible values of the ParameterId_t parameterId.
- A range of $2^{15}$ values is reserved for protocol-defined parameters. All the parameter_id values defined by the 2.5 version of the protocol and all future revisions of the same major version must use values in this range.
- A range of $2^{15}$ values is reserved for vendor-defined parameters. The 2.5 version of the protocol and any future revisions of the protocol that correspond to the same major version are not allowed to use values in this range.
- The maximum length of any parameter is limited to $2^{16}$ octets.

Subject to the above constraints, different PSMs might choose different representations for the ParameterId_t. For example, a PSM could represent parameterId using short integers while another PSM may use strings.

8.3.5.10 Count

Count is used by several Submessages and enables a receiver to detect duplicates of the same Submessage.
Table 8.29 - Structure of the Count SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Count_t</td>
<td>Count value</td>
</tr>
</tbody>
</table>

### 8.3.5.11 ChangeCount

**ChangeCount** is used in the Gap Submessage and enables the sender to indicate the number of changes within the Gap that belong to a certain category.

Table 8.30 - Structure of the ChangeCount SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>ChangeCount_t</td>
<td>The number of changes belonging to a category</td>
</tr>
</tbody>
</table>

### 8.3.5.12 Checksum

**Checksum** is used in the HeaderExtension Submessage and enables the receiver to detect messages corrupted by the underlying transport.

Depending on the underlying transport used to send the RTPS message, the transport may already provide sufficient guarantee that messages are not corrupted. In these cases, the Checksum may be omitted from the HeaderExtension. The specific behavior shall be defined for each Transport PSM.

Table 8.31 - Structure of the Checksum SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Checksum32_t, or Checksum64_t, or Checksum128_t</td>
<td>A checksum of the RTPS Message, including the RTPS Header.</td>
</tr>
</tbody>
</table>

### 8.3.5.13 MessageLength

**MessageLength** is used in the HeaderExtension Submessage and enables the sender to indicate the length of the RTPS message.

Depending on the underlying transport used to send the RTPS message, the length of the RTPS message may already or be derivable from the information in the transport header. In these cases, the MessageLength may be omitted from the HeaderExtension. The specific behavior shall be defined for each Transport PSM.

Table 8.32 - Structure of the MessageLength SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>MessageLength_t</td>
<td>The length of the RTPS Message, including the RTPS Header.</td>
</tr>
</tbody>
</table>

### 8.3.5.14 UExtension4

**UExtension4** is used in the HeaderExtension Submessage and enables the sender to add 4 octets of information to the header. This version of the specification does not interpret these bytes. It is intended to support future extensions of the RTPS specification.
Table 8.33 - Structure of the UExtension4 SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>UExtension4_t</td>
<td>Undefined. Intended for a future extension.</td>
</tr>
</tbody>
</table>

8.3.5.15 WExtension8

WExtension8 is used in the HeaderExtension Submessage and enables the sender to add 8 octets of information to the header. This version of the specification does not interpret these bytes. It is intended to support future extensions of the RTPS specification.

Table 8.34 - Structure of the WExtension8 SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>WExtension8_t</td>
<td>Undefined. Intended for a future extension.</td>
</tr>
</tbody>
</table>

8.3.5.16 LocatorList

LocatorList is used to specify a list of locators.

Table 8.35 - Structure of the LocatorList SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Locator_t[*]</td>
<td>List of locators</td>
</tr>
</tbody>
</table>

8.3.5.17 SerializedData

SerializedData contains the serialized representation of the value of a data-object. The RTPS protocol does not interpret the serialized data-stream. Therefore, it is represented as opaque data. For additional information see, 10 Serialized Payload Representation.

Table 8.36 – Structure of the SerializedData SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>octet[*]</td>
<td>Serialized data-stream</td>
</tr>
</tbody>
</table>

8.3.5.18 SerializedDataFragment

SerializedDataFragment contains the serialized representation of a data-object that has been fragmented. Like for unfragmented SerializedData, the RTPS protocol does not interpret the fragmented serialized data-stream. Therefore, it is represented as opaque data. For additional information see, Serialized Payload Representation.

Table 8.37 - SerializedDataFragment

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>octet[*]</td>
<td>Serialized data-stream fragment</td>
</tr>
</tbody>
</table>

8.3.5.19 GroupDigest

GroupDigest is used to communicate a set of EntityId_t in a compact manner.
Table 8.38 - Structure of the GroupDigest SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>GroupDigest_t</td>
<td>Type used to hold a digest value that uniquely identifies a group of Entities belonging to the same Participant.</td>
</tr>
</tbody>
</table>

8.3.6 The RTPS Header

As described in 8.3.3, every RTPS Message must start with a Header.

8.3.6.1 Purpose

The Header is used to identify the message as belonging to the RTPS protocol, to identify the version of the RTPS protocol used, and to provide context information that applies to the Submessages contained within the message.

8.3.6.2 Content

The elements that form the structure of the Header were described in 8.3.3.1. The structure of the Header can only be changed if the major version of the protocol is also changed.

8.3.6.3 Validity

A Header is invalid when any of the following are true:

- The Message has less than the required number of octets to contain a full Header. The number required is defined by the PSM.
- Its protocol value does not match the value of PROTOCOL_RTPS².
- The major protocol version is larger than the major protocol version supported by the implementation.

8.3.6.4 Change in state of Receiver

The initial state of the Receiver is described in 8.3.4. This sub clause describes how the Header of a new Message affects the state of the Receiver.

Receiver.sourceGuidPrefix = Header.guidPrefix
Receiver.sourceVersion = Header.version
Receiver.sourceVendorId = Header.vendorId
Receiver.haveTimestamp = false

8.3.6.5 Logical Interpretation

None

8.3.7 The RTPS HeaderExtension

8.3.7.1 Purpose

The HeaderExtension is used to convey optional information about the RTPS Message. This submessage, if present, shall appear immediately after the RTPS Header.

8.3.7.2 Content

The elements that form the structure of the HeaderExtension were described in 8.3.3.2.

² The actual value of the PROTOCOL_RTPS constant is provided by the PSM.
8.3.7.3 Validity

This **HeaderExtension** is invalid when any of the following is true:

- The submessage is present but does not immediately follow the RTPS **Header**.
- The `submessageLength` in the **Submessage** header is too small to fit the fields that are present as indicated by the submessage flags.
- The `messageLength` is too small to fit the RTPS **Header** and the **HeaderExtension**.
- The parameters are malformed.

8.3.7.4 Change in state of Receiver

The state of the **Receiver** changes is certain fields are present in the **HeaderExtension** as indicated below:

```plaintext
IF ( LengthFlag ) {  
    Receiver.messageLength   = HeaderExtension.messageLength 
} 

IF ( TimestampFlag ) {  
    Receiver.rtpsSendTimestamp = HeaderExtension.rtpsSendTimestamp  
    Receiver.rtpsReceptionTimestamp = GetCurrentTime()  
    RECEIVER.clockSkewDetected =  
        CLOCK_SKEW_DETECTED(receptionTime, Receiver.rtpsSendTimestamp)  
} 

IF ( uExtensionFlag ) {  
    Receiver.uExtension4 = HeaderExtension.uExtension4  
} 

IF ( wExtensionFlag ) {  
    Receiver.wExtension8 = HeaderExtension.wExtension8  
} 

IF ( ChecksumFlags != 00 ) {  
    Receiver.messageChecksum = HeaderExtension.messageChecksum  
} 

IF ( ParametersFlag) {  
    Receiver.parameters = HeaderExtension.parameters  
}
```

8.3.7.5 Logical Interpretation

The **HeaderExtension** may be sent to communicate the length of the RTPS **Message**, the time when the message was sent, a checksum of the RTPS **Message**, or additional information about the RTPS message.

The `messageLength` may be useful for managing memory while receiving incoming RTPS messages. The value of the `messageLength` shall indicate the length of the entire RTPS **Message** starting from the beginning of the RTPS **Header**.

The `rtpsSendTimestamp` represents the time the RTPS message was sent. It is measured using the clock of the sending Participant.

The `rtpsReceptionTimestamp` represents the time when the RTPS message was received. It is measured using the clock of the receiving Participant.

The function `CLOCK_SKEW_DETECTED()` represents an implementation-specific function used to determine whether the sending and receiving participant clocks are not synchronized or have too large a skew. The criteria
used is implementation-specific, but it could be as simple as detecting a message delay that exceeds a pre-configured threshold.

The value of clockSkewDetected represents whether clock skew was detected for this RTPS message. This can be used to correct the source timestamps associated with the DATA messages. See 8.3.8.11.5

The uExtension4 and wExtension8 may be useful to communicate additional data on the header in a future version of the RTPS specification.

The messageChecksum may be useful to detect message corruption by the underlying transport. It shall be computed over the entire RTPS Message. For the purpose of computing the checksum, the value of the messageChecksum field in the RTPS HeaderExtension shall be set to zero.

The parameters contain additional information about the RTPS Message. This is intended for future extensibility and vendor extensions. This version of the specification does not define any parameters in the HeaderExtension.

8.3.8 RTPS Submessages

The RTPS protocol version 2.5 defines several kinds of Submessages. They are categorized into two groups: Entity- Submessages and Interpreter-Submessages. Entity Submessages target an RTPS Entity. Interpreter Submessages modify the RTPS Receiver state and provide context that helps process subsequent Entity Submessages.

The Entity Submessages are:

- **Data**: Contains information regarding the value of an application Date-object. Data Submessages are sent by Writers to Readers.
- **DataFrag**: Equivalent to Data, but only contains a part of the new value (one or more fragments). Allows data to be transmitted as multiple fragments to overcome transport message size limitations.
- **Heartbeat**: Describes the information that is available in a Writer. Heartbeat messages are sent by a Writer to one or more Readers.
- **HeartbeatFrag**: For fragmented data, describes what fragments are available in a Writer. HeartbeatFrag messages are sent by a Writer to one or more Readers.
- **Gap**: Describes the information that is no longer relevant to Readers. Gap messages are sent by a Writer to one or more Readers.
- **AckNack**: Provides information on the state of a Reader to a Writer. AckNack messages are sent by a Reader to one or more Writers.
- **NackFrag**: Provides information on the state of a Reader to a Writer, more specifically what fragments the Reader is still missing. NackFrag messages are sent by a Reader to one or more Writers.

The Interpreter Submessages are:

- **HeaderExtension**: Provides additional information that logically belongs in the RTPS Header. The additional information is included inside this submessage, instead of the RTPS Header, in order to preserve interoperability with earlier versions of the RTPS protocol. RTPS version 2.4 and earlier version are not able to process the HeaderExtension and will skip this submessage.
- **InfoSource**: Provides information about the source from which subsequent Entity Submessages originated. This Submessage is primarily used for relaying RTPS Submessages. This is not discussed in the current specification.
- **InfoDestination**: Provides information about the final destination of subsequent Entity Submessages. This Submessage is primarily used for relaying RTPS Submessages. This is not discussed in the current specification.
- **InfoReply**: Provides information about where to reply to the entities that appear in subsequent Submessages.
- **InfoTimestamp**: Provides a source timestamp for subsequent Entity Submessages.
- **Pad**: Used to add padding to a Message if needed for memory alignment.

Figure 8.14 - RTPS Submessages

This sub clause describes each of the Submessages and their interpretation. Each Submessage is described in the same manner under the headings described in Table 8.39.

Table 8.39 – Scheme used to describe each Submessage

<table>
<thead>
<tr>
<th>heading</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>High-level description of the main purpose of the Submessage</td>
</tr>
<tr>
<td>Content</td>
<td>Description of the <strong>SubmessageHeader</strong> (<strong>SubmessageId</strong> and <strong>flags</strong>). Description of the <strong>SubmessageElements</strong> that can appear in the Submessage.</td>
</tr>
<tr>
<td>Validity</td>
<td>Constraints that must be met by the Submessage in order for it to be valid.</td>
</tr>
<tr>
<td>Change in State of the Receiver</td>
<td>The interpretation and meaning of a Submessage within a Message may depend on the previous Submessages within that same Message. As described in 8.3.4 this context is modeled as the state of a Receiver object.</td>
</tr>
<tr>
<td>Logical interpretation</td>
<td>Description of how the Submessage should be interpreted.</td>
</tr>
</tbody>
</table>
8.3.8.1 AckNack

8.3.8.1.1 Purpose

This Submessage is used to communicate the state of a Reader to a Writer. The Submessage allows the Reader to inform the Writer about the sequence numbers it has received and which ones it is still missing. This Submessage can be used to do both positive and negative acknowledgments.

8.3.8.1.2 Content

The elements that form the structure of the AckNack message are described in the table below.

<table>
<thead>
<tr>
<th>element</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EndiannessFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates endianness.</td>
</tr>
<tr>
<td>FinalFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates to the Writer whether a response is mandatory.</td>
</tr>
<tr>
<td>readerId</td>
<td>EntityId</td>
<td>Identifies the Reader entity that acknowledges receipt of certain sequence numbers and/or requests to receive certain sequence numbers.</td>
</tr>
<tr>
<td>writerId</td>
<td>EntityId</td>
<td>Identifies the Writer entity that is the target of the AckNack message. This is the Writer Entity that is being asked to re-send some sequence numbers or is being informed of the reception of certain sequence numbers.</td>
</tr>
<tr>
<td>readerSNState</td>
<td>SequenceNumberSet</td>
<td>Communicates the state of the reader to the writer. All sequence numbers up to the one prior to readerSNState.base are confirmed as received by the reader. The sequence numbers that appear in the set indicate missing sequence numbers on the reader side. The ones that do not appear in the set are undetermined (could be received or not).</td>
</tr>
<tr>
<td>count</td>
<td>Count</td>
<td>A counter that is incremented each time a new AckNack message is sent. Provides the means for a Writer to detect duplicate AckNack messages that can result from the presence of redundant communication paths.</td>
</tr>
</tbody>
</table>

8.3.8.1.3 Validity

This Submessage is invalid when any of the following is true:

- `submessageLength` in the Submessage header is too small.
- `readerSNState` is invalid (as defined in Section 8.3.5.5).

8.3.8.1.4 Change in state of Receiver

None

8.3.8.1.5 Logical Interpretation

The Reader sends the AckNack message to the Writer to communicate its state with respect to the sequence numbers used by the Writer.

The Writer is uniquely identified by its GUID. The Writer GUID is obtained using the state of the Receiver:

\[
\text{writerGUID} = \{ \text{Receiver.destGuidPrefix}, \text{AckNack.writerId} \}
\]
The Reader is uniquely identified by its GUID. The Reader GUID is obtained using the state of the Receiver:
\[ \text{readerGUID} = \{ \text{Receiver.sourceGuidPrefix, AckNack.readerId} \} \]

The message serves two purposes simultaneously:
- The Submessage acknowledges all sequence numbers up to and including the one just before the lowest sequence number in the SequenceNumberSet (that is readerSNState.base -1).
- The Submessage negatively-acknowledges (requests) the sequence numbers that appear explicitly in the set.

The mechanism to explicitly represent sequence numbers depends on the PSM. Typically, a compact representation (such as a bitmap) is used.

The FinalFlag indicates whether a Heartbeat by the Writer is expected by the Reader or if the decision is left to the Writer. The use of this flag is described in Section 8.4.

### 8.3.8.2 Data

This Submessage is sent from an RTPS Writer to an RTPS Reader.

#### 8.3.8.2.1 Purpose

The Submessage notifies the RTPS Reader of a change to a data-object belonging to the RTPS Writer. The possible changes include both changes in value as well as changes to the lifecycle of the data-object.

#### 8.3.8.2.2 Contents

The elements that form the structure of the Data message are described in the table below.

<table>
<thead>
<tr>
<th>element</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EndiannessFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates endianness.</td>
</tr>
<tr>
<td>InlineQosFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates to the Reader the presence of a ParameterList containing QoS parameters that should be used to interpret the message.</td>
</tr>
<tr>
<td>DataFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates to the Reader that the dataPayload submessage element contains the serialized value of the data-object.</td>
</tr>
<tr>
<td>KeyFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates to the Reader that the dataPayload submessage element contains the serialized value of the key of the data-object.</td>
</tr>
<tr>
<td>NonStandardPayloadFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates to the Reader that the serializedPayload submessage element is not formatted according to Section 10.</td>
</tr>
<tr>
<td>readerId</td>
<td>EntityId</td>
<td>Identifies the RTPS Reader entity that is being informed of the change to the data-object.</td>
</tr>
<tr>
<td>writerId</td>
<td>EntityId</td>
<td>Identifies the RTPS Writer entity that made the change to the data-object.</td>
</tr>
</tbody>
</table>
### 8.3.8.2.3 Validity

This Submessage is invalid when any of the following is true:
- `submessageLength` in the Submessage header is too small.
- `writerSN.value` is not strictly positive (1, 2, ...) or is `SEQUENCENUMBER_UNKNOWN`.
- `inlineQos` is invalid.

### 8.3.8.2.4 Change in state of Receiver

None

### 8.3.8.2.5 Logical Interpretation

The RTPS Writer sends the Data Submessage to the RTPS Reader to communicate changes to the data-objects within the writer. Changes include both changes in value as well as changes to the lifecycle of the data-object.

Changes to the value are communicated by the presence of the `serializedPayload`. When present, the `serializedPayload` is interpreted either as the value of the data-object or as the key that uniquely identifies the data-object from the set of registered objects.
- If the `DataFlag` is set and the `KeyFlag` is not set, the `serializedPayload` element is interpreted as the value of the data-object.
- If the `KeyFlag` is set and the `DataFlag` is not set, the `serializedPayload` element is interpreted as the value of the key that identifies the registered instance of the data-object.

If the `InlineQosFlag` is set, the `inlineQos` element contains QoS values that override those of the RTPS Writer and should be used to process the update. For a complete list of possible in-line QoS parameters, see Table 8.85.

If the `NonStandardPayloadFlag` is set then the `serializedPayload` element is not formatted according to Section 10. This flag is informational. It indicates that the SerializedPayload has been transformed as described in another specification. For example, this flag should be set when the SerializedPayload is transformed as described in the DDS-Security specification.

The Writer is uniquely identified by its GUID. The Writer GUID is obtained using the state of the Receiver:

```plaintext
writerGUID = { Receiver.sourceGuidPrefix, Data.writerId }
```

The Reader is uniquely identified by its GUID. The Reader GUID is obtained using the state of the Receiver:

```plaintext
readerGUID = { Receiver.destGuidPrefix, Data.readerId }
```

The Data.readerId can be ENTITYID_UNKNOWN, in which case the Data applies to all Readers of that writerGUID within the Participant identified by the GuidPrefix_t Receiver.destGuidPrefix.
8.3.8.3 DataFrag

This Submessage is sent from an RTPS **Writer** to an RTPS **Reader**.

8.3.8.3.1 Purpose

The **DataFrag** Submessage extends the **Data** Submessage by enabling the `serializedData` to be fragmented and sent as multiple **DataFrag** Submessages. The fragments contained in the **DataFrag** Submessages are then re-assembled by the RTPS **Reader**.

Defining a separate **DataFrag** Submessage in addition to the **Data** Submessage, offers the following advantages:

- It keeps variations in contents and structure of each Submessage to a minimum. This enables more efficient implementations of the protocol as the parsing of network packets is simplified.
- It avoids having to add fragmentation information as in-line QoS parameters in the **Data** Submessage. This may not only slow down performance, it also makes on-the-wire debugging more difficult, as it is no longer obvious whether data is fragmented or not and which message contains what fragment(s).

8.3.8.3.2 Contents

The elements that form the structure of the **DataFrag** Submessage are described in the table below.

**Table 8.42 – Structure of the DataFrag Submessage**

<table>
<thead>
<tr>
<th>element</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EndiannessFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates endianness.</td>
</tr>
<tr>
<td>InlineQosFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates to the Reader the presence of a ParameterList containing QoS parameters that should be used to interpret the message.</td>
</tr>
<tr>
<td>NonStandardPayloadFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates to the Reader that the serializedPayload submessage element is not formatted according to Section 10.</td>
</tr>
<tr>
<td>KeyFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates to the Reader that the dataPayload submessage element contains the serialized value of the key of the data-object.</td>
</tr>
<tr>
<td>readerId</td>
<td>EntityId</td>
<td>Identifies the RTPS <strong>Reader</strong> entity that is being informed of the change to the data-object.</td>
</tr>
<tr>
<td>writerId</td>
<td>EntityId</td>
<td>Identifies the RTPS <strong>Writer</strong> entity that made the change to the data-object.</td>
</tr>
<tr>
<td>writerSN</td>
<td>SequenceNumber</td>
<td>Uniquely identifies the change and the relative order for all changes made by the RTPS <strong>Writer</strong> identified by the writerGuid. Each change gets a consecutive sequence number. Each RTPS <strong>Writer</strong> maintains its own sequence number.</td>
</tr>
<tr>
<td>fragmentStartingNum</td>
<td>FragmentNumber</td>
<td>Indicates the starting fragment for the series of fragments in <code>serializedData</code>. Fragment numbering starts with number 1.</td>
</tr>
<tr>
<td>Field</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>fragmentsInSubmessage</td>
<td>ushort</td>
<td>The number of consecutive fragments contained in this Submessage, starting at fragmentStartingNum.</td>
</tr>
<tr>
<td>dataSize</td>
<td>ulong</td>
<td>The total size in bytes of the original data before fragmentation.</td>
</tr>
<tr>
<td>fragmentSize</td>
<td>ushort</td>
<td>The size of an individual fragment in bytes. The maximum fragment size equals 64K.</td>
</tr>
<tr>
<td>inlineQos</td>
<td>ParameterList</td>
<td>Present only if the InlineQosFlag is set in the header. Contains QoS that may affect the interpretation of the message.</td>
</tr>
<tr>
<td>serializedPayload</td>
<td>SerializedPayload</td>
<td>A consecutive series of fragments, starting at fragmentStartingNum for a total of fragmentsInSubmessage. Represents part of the new value of the data-object after the change.</td>
</tr>
</tbody>
</table>

8.3.8.3.3 Validity

This Submessage is **invalid** when any of the following is true:

- `submessageLength` in the Submessage header is too small.
- `writerSN.value` is not strictly positive (1, 2, ...) or is `SEQUENCENUMBER_UNKNOWN`.
- `fragmentStartingNum.value` is not strictly positive (1, 2, ...) or exceeds the total number of fragments (see below).
- `fragmentSize` exceeds `dataSize`.
- The size of `serializedData` exceeds `fragmentsInSubmessage * fragmentSize`.
- `inlineQos` is invalid.

8.3.8.3.4 Change in state of Receiver

None

8.3.8.3.5 Logical Interpretation

The `DataFrag` Submessage extends the `Data` Submessage by enabling the `serializedData` to be fragmented and sent as multiple `DataFrag` Submessages. Once the `serializedData` is re-assembled by the RTPS Reader, the interpretation of the `DataFrag` Submessages is identical to that of the `Data` Submessage.

How to re-assemble `serializedData` using the information in the `DataFrag` Submessage is described below.

The total size of the data to be re-assembled is given by `dataSize`. Each `DataFrag` Submessage contains a contiguous segment of this data in its `serializedData` element. The size of the segment is determined by the `size` of the `serializedData` element. During re-assembly, the offset of each segment is determined by:

\[(\text{fragmentStartingNum} - 1) \times \text{fragmentSize}\]
The data is fully re-assembled when all fragments have been received. The total number of fragments to expect equals:

\[(\text{dataSize} / \text{fragmentSize}) + ((\text{dataSize} \% \text{fragmentSize}) \ ? 1 : 0)\]

Note that each DataFrag Submessage may contain multiple fragments. An RTPS Writer will select fragmentSize based on the smallest message size supported across all underlying transports. If some RTPS Readers can be reached across a transport that supports larger messages, the RTPS Writer can pack multiple fragments into a single DataFrag Submessage or may even send a regular Data Submessage if fragmentation is no longer required. For more details, see 8.4.14.1.

When sending inlineQos with DataFrag Submessages, it is only required to send the inlineQos with the first DataFrag Submessage for a given Writer sequence number. Sending the same inlineQos with every DataFrag Submessage for a given Writer sequence number is redundant.

### 8.3.8.4 Gap

#### 8.3.8.4.1 Purpose

This Submessage is sent from an RTPS Writer to an RTPS Reader and indicates to the RTPS Reader that a set of changes in the RTPS Writer HistoryCache are not available to one or more RTPS Reader endpoints and will not be sent to them. The changes are identified by their corresponding sequence numbers. The set may contain a contiguous range of sequence numbers as well as a list of sequence numbers beyond the range.

In addition, the Gap message may also inform the RTPS Reader of the number of changes in the Gap representing changes filtered for that RTPS Reader. These “filtered changes” are changes still present in the Writer HistoryCache that will not be sent to the RTPS Reader due to the application of a filtering criteria by the Writer. Examples of filtering criteria include:

- A content filter specified by the RTPS Reader.
- A time filter specified by the RTPS Reader.
- A writer-side specification that the sample is written directed to a specific set of RTPS Reader endpoints that do not include the RTPS Reader that receives the Gap.

Filtered changes should not be treated as samples lost by the RTPS Reader.

#### 8.3.8.4.2 Content

The elements that form the structure of the Gap message are described in the table below.

<table>
<thead>
<tr>
<th>element</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EndiannessFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates endianness.</td>
</tr>
<tr>
<td>GroupInfoFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates the presence of additional information about the group of writers (Writer Group) the sender belongs to.</td>
</tr>
<tr>
<td>FilteredCountFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates the presence of the filteredCount submessage element.</td>
</tr>
<tr>
<td>readerId</td>
<td>EntityId</td>
<td>Identifies the Reader Entity that is being informed of the set of changes that are not available. The readerId may be set to ENTITYID_UNKNOWN to indicate the Gap message applies to all Readers within the receiving Participant that are matched with the Writer specified by the writerId.</td>
</tr>
<tr>
<td>writerId</td>
<td>EntityId</td>
<td>Identifies the Writer Entity to which the not-available changes apply.</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>gapStart</td>
<td>SequenceNumber</td>
<td>Identifies the first sequence number in the interval of sequence numbers corresponding to the set of not-available changes.</td>
</tr>
</tbody>
</table>
| gapList  | SequenceNumberSet | Serves two purposes:  
(1) Identifies the last sequence number in the interval of sequence numbers corresponding to the set of not-available changes.  
(2) Identifies an additional list of sequence numbers corresponding to not-available changes. |
| gapStartGSN | SequenceNumber | Present only if the GroupInfoFlag is set in the header.  
Identifies the group sequence number corresponding to the sample identified by gapStart. |
| gapEndGSN | SequenceNumber | Present only if the GroupInfoFlag is set in the header.  
Identifies the end of a continuous range of GSNs starting at gapStartGSN that are not available to the Reader.  
It shall be greater than or equal to the group sequence number corresponding to the sample identified by gapList.bitmapBase. |
| filteredCount | ChangeCount | Present only if the FilteredCountFlag is set in the header.  
Indicates the number of changes corresponding to the set of sequence numbers in the Gap that are still present in the Writer cache but are being filtered for the RTPS Reader. |

### 8.3.8.4.3 Validity

This Submessage is invalid when any of the following is true:
- submessageLength in the Submessage header is too small.
- gapStart is zero or negative.
- gapList is invalid (as defined in 8.3.5.5).

If GroupInfoFlag is set and:
- gapStartGSN.value is zero or negative  
- gapEndGSN.value is zero or negative  
- gapEndGSN.value < gapStartGSN.value - 1

### 8.3.8.4 Change in state of Receiver

None

### 8.3.8.5 Logical Interpretation

The RTPS Writer sends the Gap message to the RTPS Reader to communicate that certain changes are not available to the Reader. This is typically caused by Writer-side filtering of the sample (content-filtered topics, time-based filtering) as well as samples being replaced with new samples in a Writer with HISTORY Qos KEEP_LAST.

The set of sequence numbers that identify the corresponding changes appear in the Gap message divided in two groups:

1. All sequence numbers in the range gapStart <= sequence_number <= gapList.base - 1
2. All the sequence numbers that appear explicitly listed in the gapList.
This set will be referred to as the Gap::sequence_number_set.

The Writer is uniquely identified by its GUID. The Writer GUID is obtained using the state of the Receiver:
$$\text{writerGUID} = \{ \text{Receiver.sourceGuidPrefix}, \text{Gap.writerId} \}$$

The Reader is uniquely identified by its GUID. The Reader GUID is obtained using the state of the Receiver:
$$\text{readerGUID} = \{ \text{Receiver.destGuidPrefix}, \text{Gap.readerId} \}$$

The Gap readerId can be ENTITYID_UNKNOWN, in which case the Gap applies to all Readers of that writerGUID within the Participant.

The Writer sets the GroupInfoFlag to indicate the presence of the gapStartGSN and gapEndGSN elements. These fields provide information related to the CacheChanges of Writers belonging to a Writer Group. See section 8.7.6 for how DDS uses this feature.

The gapEndGSN can extend past the Group Sequence Number that corresponds to gapList.bitmapBase in situations where those additional Group Sequence Numbers have been written by other Writers.

The filteredCount is the count of the sequence numbers in the Gap::sequence_number_set that correspond to changes that are still in the RTPS Writer HistoryCache but the Writer will not send as they are considered not relevant to the Reader receiving the Gap. This means the Reader would not have been interested in them because they do not meet some reader-specified criteria or because they were written exclusively targeting other Reader endpoints.

The filteredCount, if present, shall include:

- Changes filtered due to the writer-side application of a content-filtered-topic specified by the Reader.
- Changes filtered due to the writer-side application a time filter specified by the Reader.
- Changes that are written in ways that intend to exclude delivery to the Reader.
- Changes that are referenced in a Gap as non-relevant for a Reader may still be in the Writer HistoryCache. They may be relevant to other Readers (e.g. a reader without a content filter).

The filteredCount shall not include changes that are no longer in the RTPS Writer HistoryCache unless the RTPS Writer knows that the sample would have been filtered for the Reader even if it was still in the cache. In particular, the filteredCount does not include changes that correspond to samples replaced by HISTORY Qos set to KEEP_LAST as well as samples removed from the Writer cache due to hitting any resource limit.

### 8.3.8.5 HeaderExtension

This submessage is logically part of the header and it is defined in 8.3.7.

### 8.3.8.6 Heartbeat

#### 8.3.8.6.1 Purpose

This message is sent from an RTPS Writer to an RTPS Reader to communicate the sequence numbers of changes that the Writer has available.

#### 8.3.8.6.2 Content

The elements that form the structure of the Heartbeat message are described in the table below.
Table 8.44 - Structure of the Heartbeat Submessage

<table>
<thead>
<tr>
<th>element</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EndiannessFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates endianness.</td>
</tr>
<tr>
<td>FinalFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates whether the Reader is required to respond to the Heartbeat or if it is just an advisory heartbeat.</td>
</tr>
<tr>
<td>LivelinessFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates that the DDS DataWriter associated with the RTPS Writer of the message has manually asserted its LIVELINESS.</td>
</tr>
<tr>
<td>GroupInfoFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates the presence of additional information about the group of writers (Writer Group) the sender belongs to.</td>
</tr>
<tr>
<td>readerId</td>
<td>EntityId</td>
<td>Identifies the Reader Entity that is being informed of the availability of a set of sequence numbers. Can be set to ENTITYID_UNKNOWN to indicate all readers for the writer that sent the message.</td>
</tr>
<tr>
<td>writerId</td>
<td>EntityId</td>
<td>Identifies the Writer Entity to which the range of sequence numbers applies.</td>
</tr>
<tr>
<td>firstSN</td>
<td>SequenceNumber</td>
<td>If samples are available in the Writer, identifies the first (lowest) sequence number that is available in the Writer. If no samples are available in the Writer, identifies the lowest sequence number that is yet to be written by the Writer.</td>
</tr>
<tr>
<td>lastSN</td>
<td>SequenceNumber</td>
<td>Identifies the last (highest) sequence number that the Writer has ever written.</td>
</tr>
<tr>
<td>count</td>
<td>Count</td>
<td>A counter that is incremented each time a new Heartbeat message is sent. Provides the means for a Reader to detect duplicate Heartbeat messages that can result from the presence of redundant communication paths.</td>
</tr>
<tr>
<td>currentGSN</td>
<td>SequenceNumber</td>
<td>Present only if the GroupInfoFlag is set in the header. Identifies the last (highest) group sequence number written by any DataWriter in the Writer’s Group at the time that the HeartBeat was sent.</td>
</tr>
<tr>
<td>firstGSN</td>
<td>SequenceNumber</td>
<td>Present only if the GroupInfoFlag is set in the header. Identifies the group sequence number corresponding to the sample identified by sequence number firstSN.</td>
</tr>
<tr>
<td>lastGSN</td>
<td>SequenceNumber</td>
<td>Present only if the GroupInfoFlag is set in the header. Identifies the group sequence number corresponding to the sample identified by sequence number lastSN.</td>
</tr>
<tr>
<td>writerSet</td>
<td>GroupDigest</td>
<td>Present only if the GroupInfoFlag is set in the header. Identifies the subset of Writers that belong to the Writer’s Group at the time the sample with currentGSN was written.</td>
</tr>
<tr>
<td>secureWriterSet</td>
<td>GroupDigest</td>
<td>Present only if the GroupInfoFlag is set in the header. Reserved for use by the DDS-Security Specification.</td>
</tr>
</tbody>
</table>
The following examples illustrate how the `firstSN.value` and `lastSN.value` are assigned in various scenarios.

**Example 1.** A **Writer** that has never written any samples before sending a **Heartbeat** will send a **Heartbeat** with `firstSN.value = 1, lastSN.value = 0`.

**Example 2.** A **Writer** that has only one sample in its cache with sequence number `SN` will send a **Heartbeat** with `firstSN.value = lastSN.value = SN`.

**Example 3.** A **Writer** that has written 10 samples and still has the last 5 samples in its cache will send a **Heartbeat** with `firstSN.value = 6, lastSN.value = 10`.

**Example 4.** A **Writer** that has written 10 samples before sending a **Heartbeat** but does not have any samples available at the time of the **Heartbeat** will send a **Heartbeat** with `firstSN.value = 11, lastSN.value = 10`.

### 8.3.8.6.3 Validity

This Submessage is **invalid** when any of the following is true:

- `submessageLength` in the Submessage header is too small
- `firstSN.value` is zero or negative
- `lastSN.value` is negative
- `lastSN.value < firstSN.value - 1`

If `GroupInfoFlag` is set and:

- `currentGSN.value` is zero or negative
- `firstGSN.value` is zero or negative
- `lastGSN.value` is negative
- `lastGSN.value < firstGSN.value - 1`
- `currentGSN.value < firstGSN.value`
- `currentGSN.value > lastGSN.value`

### 8.3.8.6.4 Change in state of Receiver

None

### 8.3.8.6.5 Logical Interpretation

The **Heartbeat** message serves two purposes:

1. It informs the **Reader** of the sequence numbers that are available in the writer’s **HistoryCache** so that the **Reader** may request (using an **AckNack**) any that it has missed.
2. It requests the **Reader** to send an acknowledgement for the **CacheChange** changes that have been entered into the reader’s **HistoryCache** such that the **Writer** knows the state of the reader.

All **Heartbeat** messages serve the first purpose. That is, the **Reader** will always find out the state of the writer’s **HistoryCache** and may request what it has missed. Normally, the RTPS **Reader** would only send an AckNack message if it is missing a **CacheChange**.

The **Writer** uses the **FinalFlag** to request the **Reader** to send an acknowledgment for the sequence numbers it has received. If the **Heartbeat** has the **FinalFlag** set, then the **Reader** is not required to send an **AckNack** message back. However, if the **FinalFlag** is not set, then the **Reader** must send an **AckNack** message indicating which **CacheChange** changes it has received, even if the **AckNack** indicates it has received all **CacheChange** changes in the writer’s **HistoryCache**.

The **Writer** sets the **LivelinessFlag** to indicate that the DDS DataWriter associated with the RTPS **Writer** of the message has manually asserted its liveness using the appropriate DDS operation (see the DDS Specification). The RTPS **Reader** should therefore renew the manual liveness lease of the corresponding remote DDS DataWriter.
The Writer sets the GroupInfoFlag to indicate the presence of the currentGSN, firstGSN, lastGSN, writerSet, and secureWriterSet elements. These fields provide relate the CacheChanges of Writers belonging to a Writer Group. See 8.7.6 for how DDS uses this feature.

The Writer is identified uniquely by its GUID. The Writer GUID is obtained using the state of the Receiver:

\[
\text{writerGUID} = \{ \text{Receiver.sourceGuidPrefix}, \text{Heartbeat.writerId} \}
\]

The Reader is identified uniquely by its GUID. The Reader GUID is obtained using the state of the Receiver:

\[
\text{readerGUID} = \{ \text{Receiver.destGuidPrefix}, \text{Heartbeat.readerId} \}
\]

The Heartbeat.readerId can be ENTITYID_UNKNOWN, in which case the Heartbeat applies to all Readers of that writerGUID within the Participant.

8.3.8.7 HeartbeatFrag

8.3.8.7.1 Purpose

When fragmenting data and until all fragments are available, the HeartbeatFrag Submessage is sent from an RTPS Writer to an RTPS Reader to communicate which fragments the Writer has available. This enables reliable communication at the fragment level.

Once all fragments are available, a regular Heartbeat message is used.

8.3.8.7.2 Content

The elements that form the structure of the HeartbeatFrag message are described in the table below.

<table>
<thead>
<tr>
<th>element</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EndiannessFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates endianness.</td>
</tr>
<tr>
<td>readerId</td>
<td>EntityId</td>
<td>Identifies the Reader Entity that is being informed of the availability of fragments. Can be set to ENTITYID_UNKNOWN to indicate all readers for the writer that sent the message.</td>
</tr>
<tr>
<td>writerId</td>
<td>EntityId</td>
<td>Identifies the Writer Entity that sent the Submessage.</td>
</tr>
<tr>
<td>writerSN</td>
<td>SequenceNumber</td>
<td>Identifies the sequence number of the data change for which fragments are available.</td>
</tr>
<tr>
<td>lastFragmentNum</td>
<td>FragmentNumber</td>
<td>All fragments up to and including this last (highest) fragment are available on the Writer for the change identified by writerSN.</td>
</tr>
<tr>
<td>count</td>
<td>Count</td>
<td>A counter that is incremented each time a new HeartbeatFrag message is sent. Provides the means for a Reader to detect duplicate HeartbeatFrag messages that can result from the presence of redundant communication paths.</td>
</tr>
</tbody>
</table>

8.3.8.7.3 Validity

This Submessage is invalid when any of the following is true:

- submessageLength in the Submessage header is too small
- writerSN.value is zero or negative
- lastFragmentNum.value is zero or negative
8.3.8.7.4 Change in state of Receiver

None

8.3.8.7.5 Logical Interpretation

The **HeartbeatFrag** message serves the same purpose as a regular **Heartbeat** message, but instead of indicating the availability of a range of sequence numbers, it indicates the availability of a range of fragments for the data change with sequence number **WriterSN**.

The RTPS **Reader** will respond by sending a **NackFrag** message, but only if it is missing any of the available fragments. The **Writer** is identified uniquely by its GUID. The Writer GUID is obtained using the state of the Receiver:

```
writerGUID = { Receiver.sourceGuidPrefix, HeartbeatFrag.writerId }
```

The **Reader** is identified uniquely by its GUID. The Reader GUID is obtained using the state of the Receiver:

```
readerGUID = { Receiver.destGuidPrefix, HeartbeatFrag.readerId }
```

The HeartbeatFrag.readerId can be **ENTITYID_UNKNOWN**, in which case the **HeartbeatFrag** applies to all **Readers** of that Writer GUID within the **Participant**.

8.3.8.8 InfoDestination

8.3.8.8.1 Purpose

This message is sent from an RTPS **Writer** to an RTPS **Reader** to modify the GuidPrefix used to interpret the **Reader** entityIds appearing in the Submessages that follow.

8.3.8.8.2 Content

The elements that form the structure of the **InfoDestination** message are described in the table below.

<table>
<thead>
<tr>
<th>element</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EndiannessFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates endianness.</td>
</tr>
<tr>
<td>guidPrefix</td>
<td>GuidPrefix</td>
<td>Provides the GuidPrefix that should be used to reconstruct the GUIDs of all RTPS <strong>Reader</strong> entities whose EntityIds appears in the Submessages that follow.</td>
</tr>
</tbody>
</table>

8.3.8.8.3 Validity

This Submessage is **invalid** when any of the following is true:

- **submessageLength** in the Submessage header is too small.

8.3.8.8.4 Change in state of Receiver

```
if (InfoDestination.guidPrefix != GUIDPREFIX_UNKNOWN) {
    Receiver.destGuidPrefix = InfoDestination.guidPrefix
} else {
    Receiver.destGuidPrefix = <GuidPrefix_t of the Participant receiving the_message>
}
```

8.3.8.8.5 Logical Interpretation

None
8.3.8.9  InfoReply

8.3.8.9.1  Purpose

This message is sent from an RTPS Reader to an RTPS Writer. It contains explicit information on where to send a reply to the Submessages that follow within the same message.

8.3.8.9.2  Content

The elements that form the structure of the InfoReply message are described in the table below.

Table 8.47 - Structure of the InfoReply Submessage

<table>
<thead>
<tr>
<th>element</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EndiannessFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates endianness.</td>
</tr>
<tr>
<td>MulticastFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates whether the Submessage also contains a multicast address.</td>
</tr>
<tr>
<td>unicastLocatorList</td>
<td>LocatorList</td>
<td>Indicates an alternative set of unicast addresses that the Writer should use to reach the Readers when replying to the Submessages that follow.</td>
</tr>
<tr>
<td>multicastLocatorList</td>
<td>LocatorList</td>
<td>Indicates an alternative set of multicast addresses that the Writer should use to reach the Readers when replying to the Submessages that follow. Only present when the MulticastFlag is set.</td>
</tr>
</tbody>
</table>

8.3.8.9.3  Validity

This Submessage is invalid when any of the following is true:
- `submessageLength` in the Submessage header is too small.

8.3.8.9.4  Change in state of Receiver

```plaintext
Receiver.unicastReplyLocatorList = InfoReply.unicastLocatorList if ( MulticastFlag )
{
    Receiver.multicastReplyLocatorList = InfoReply.multicastLocatorList
} else {
    Receiver.multicastReplyLocatorList = <empty>
}
```

8.3.8.9.5  Logical Interpretation

None

8.3.8.10  InfoSource

8.3.8.10.1  Purpose

This message modifies the logical source of the Submessages that follow.

8.3.8.10.2  Content

The elements that form the structure of the InfoSource message are described in the table below.
Table 8.48 - Structure of the InfoSource Submessage

<table>
<thead>
<tr>
<th>element</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EndiannessFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates endianness.</td>
</tr>
<tr>
<td>protocolVersion</td>
<td>ProtocolVersion</td>
<td>Indicates the protocol used for subsequent Submessages.</td>
</tr>
<tr>
<td>vendorId</td>
<td>VendorId</td>
<td>Indicates the VendorId of the vendor that originated the subsequent Submessages.</td>
</tr>
<tr>
<td>guidPrefix</td>
<td>GuidPrefix</td>
<td>Identifies the Participant that is the container of the RTPS Writer entities that are the source of the Submessages that follow.</td>
</tr>
</tbody>
</table>

8.3.8.10.3 Validity

This Submessage is invalid when any of the following is true:

- `submessageLength` in the Submessage header is too small.

8.3.8.10.4 Change in state of Receiver

```c
Receiver.sourceGuidPrefix = InfoSource.guidPrefix
Receiver.sourceVersion = InfoSource.protocolVersion
Receiver.sourceVendorId = InfoSource.vendorId
Receiver.unicastReplyLocatorList = { LOCATOR_INVALID }
Receiver.multicastReplyLocatorList = { LOCATOR_INVALID }
haveTimestamp = false
```

8.3.8.10.5 Logical Interpretation

None

8.3.8.11 InfoTimestamp

8.3.8.11.1 Purpose

This Submessage is used to send a timestamp which applies to the Submessages that follow within the same message.

8.3.8.11.2 Content

The elements that form the structure of the `InfoTimestamp` message are described in the table below.

Table 8.49 – Structure of the InfoTimestamp Submessage

<table>
<thead>
<tr>
<th>element</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EndiannessFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates endianness.</td>
</tr>
<tr>
<td>InvalidateFlag</td>
<td>SubmessageFlag</td>
<td>Indicates whether subsequent Submessages should be considered as having a timestamp or not.</td>
</tr>
<tr>
<td>timestamp</td>
<td>Timestamp</td>
<td>Present only if the InvalidateFlag is not set in the header. Contains the timestamp that should be used to interpret the subsequent Submessages.</td>
</tr>
</tbody>
</table>
8.3.8.11.3 Validity

This Submessage is invalid when the following is true:
  - `submessageLength` in the Submessage header is too small.

8.3.8.11.4 Change in state of Receiver

```java
if (!InfoTimestamp.InvalidateFlag) {
    Receiver.haveTimestamp = true
    Receiver.timestamp = InfoTimestamp.timestamp
    if (Receiver.clockSkewDetected) {
        Receiver.timestamp +=
            Receiver.rtpsReceptionTime - Receiver.rtpsSendTimestamp
    } else {
        Receiver.haveTimestamp = false
    }
}
```

8.3.8.11.5 Logical Interpretation

The reception of the `InfoTimestamp` submessage sets the source timestamp associated with any subsequent `Data` and `DataFrag` submessages. If applicable, the timestamp that appears in the `InfoTimestamp` is adjusted to account for clock skew between the sending and the receiving Participants.

8.3.8.12 NackFrag

8.3.8.12.1 Purpose

The NackFrag Submessage is used to communicate the state of a Reader to a Writer. When a data change is sent as a series of fragments, the NackFrag Submessage allows the Reader to inform the Writer about specific fragment numbers it is still missing.

This Submessage can only contain negative acknowledgements. Note this differs from an AckNack Submessage, which includes both positive and negative acknowledgements. The advantages of this approach include:
  - It removes the windowing limitation introduced by the AckNack Submessage. Given the size of a SequenceNumberSet is limited to 256, an AckNack Submessage is limited to NACKing only those samples whose sequence number does not exceed that of the first missing sample by more than 256. Any samples below the first missing samples are acknowledged. NackFrag Submessages on the other hand can be used to NACK any fragment numbers, even fragments more than 256 apart from those NACKed in an earlier AckNack Submessage. This becomes important when handling samples containing a large number of fragments.
  - Fragments can be negatively acknowledged in any order.

8.3.8.12.2 Content

The elements that form the structure of the NackFrag message are described in the table below.

<table>
<thead>
<tr>
<th>element</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EndiannessFlag</td>
<td>SubmessageFlag</td>
<td>Appears in the Submessage header flags. Indicates endianness.</td>
</tr>
<tr>
<td>readerId</td>
<td>EntityId</td>
<td>Identifies the Reader entity that requests to receive certain fragments.</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>writerId</td>
<td>Identifies the Writer entity that is the target of the NackFrag message. This is the Writer Entity that is being asked to re-send some fragments.</td>
<td></td>
</tr>
<tr>
<td>writerSN</td>
<td>The sequence number for which some fragments are missing.</td>
<td></td>
</tr>
<tr>
<td>fragmentNum berState</td>
<td>Communicates the state of the reader to the writer. The fragment numbers that appear in the set indicate missing fragments on the reader side. The ones that do not appear in the set are undetermined (could have been received or not).</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>A counter that is incremented each time a new NackFrag message is sent. Provides the means for a Writer to detect duplicate NackFrag messages that can result from the presence of redundant communication paths.</td>
<td></td>
</tr>
</tbody>
</table>

**8.3.8.12.3 Validity**

This Submessage is *invalid* when any of the following is true:
- `submessageLength` in the Submessage header is too small.
- `writerSN.value` is zero or negative.

`fragmentNumberState` is invalid (as defined in 8.3.5.7).

**8.3.8.12.4 Change in state of Receiver**

None

**8.3.8.12.5 Logical Interpretation**

The *Reader* sends the **NackFrag** message to the *Writer* to request fragments from the *Writer*.

The Writer is uniquely identified by its GUID. The Writer GUID is obtained using the state of the Receiver:

\[
\text{writerGUID} = \{ \text{Receiver.destGuidPrefix}, \text{NackFrag.writerId} \}
\]

The Reader is identified uniquely by its GUID. The Reader GUID is obtained using the state of the Receiver:

\[
\text{readerGUID} = \{ \text{Receiver.sourceGuidPrefix}, \text{NackFrag.readerId} \}
\]

The sequence number from which fragments are requested is given by `writerSN`. The mechanism to explicitly represent fragment numbers depends on the PSM. Typically, a compact representation (such as a bitmap) is used.

**8.3.8.13 Pad**

**8.3.8.13.1 Purpose**

The purpose of this Submessage is to allow the introduction of any padding necessary to meet any desired memory-alignment requirements. It has no other meaning.

**8.3.8.13.2 Content**

This Submessage has no contents. It accomplishes its purposes with only the Submessage header part. The amount of padding is determined by the value of `submessageLength`.

**8.3.8.13.3 Validity**

This Submessage is always valid.
8.3.8.13.4 Change in state of Receiver
None

8.3.8.13.5 Logical Interpretation
None

8.4 Behavior Module

This module describes the dynamic behavior of the RTPS entities. It describes the valid sequences of message exchanges between RTPS Writer endpoints and RTPS Reader endpoints and the timing constraints of those messages.

8.4.1 Overview

Once an RTPS Writer has been matched with an RTPS Reader, they are both responsible for ensuring that CacheChange changes that exist in the Writer’s HistoryCache are propagated to the Reader’s HistoryCache.

The Behavior Module describes how the matching RTPS Writer and Reader pair must behave in order to propagate CacheChange changes. The behavior is defined in terms of message exchanges using the RTPS Messages defined in 8.3. The Behavior Module is organized as follows:

- 8.4.2 lists what requirements all implementations of the RTPS protocol must satisfy in terms of behavior. An implementation that satisfies these requirements is considered compliant and will be interoperable with other compliant implementations.
- As implied above, it is possible for multiple implementations to satisfy the minimum requirements, where each implementation may choose a different trade-off between memory requirements, bandwidth usage, scalability, and efficiency. The RTPS specification does not mandate a single implementation with corresponding behavior. Instead, it defines the minimum requirements for interoperability and then provides two Reference Implementations, the Stateless and Stateful Reference Implementations, described in 8.4.3.
- The protocol behavior depends on such settings as the RELIABILITY QoS. 8.4.4 discusses the possible combinations.
- 8.4.5 and 8.4.6 define notational conventions and define any new types used in this module.
- 8.4.7 through 8.4.12 model the two Reference Implementations.
- 8.4.13 describes the Writer Liveliness Protocol that is used by Participants to assert the liveliness of their contained Writers.
- 8.4.14 discusses some optional behavior, including support for fragmented data.
- Finally, 8.4.15 provides guidelines for actual implementations.

Note that, as discussed earlier in 8.2.10, the Behavior Module does not model the interactions between DDS Entities and their corresponding RTPS entities. For example, it simply assumes a DDS DataWriter adds and removes CacheChange changes to and from its RTPS Writer’s HistoryCache. Changes are added by the DDS DataWriter as part of its write operation and removed when no longer needed. It is important to realize the DDS DataWriter may remove a CacheChange before it has been propagated to one or more of the matched RTPS Reader endpoints. The RTPS Writer is not in control of when a CacheChange is removed from the Writer’s HistoryCache. It is the responsibility of the DDS DataWriter to only remove those CacheChange changes that can be removed based on the communication status and the DDS DataWriter’s QoS. For example, the HISTORY QoS setting of KEEP_LAST with a depth of 1 allows a DataWriter to remove a CacheChange if a more recent change replaces the value of the same data-object.

8.4.1.1 Example Behavior

The contents of this sub clause are not part of the formal specification of the protocol. The purpose of this sub clause is to provide an intuitive understanding of the protocol.
A typical sequence illustrating the exchanges between an RTPS Writer and a matched RTPS Reader is shown in Figure 8.15. The example sequence in this case uses the Stateful Reference Implementation.

Figure 8.15 – Example Behavior

The individual interactions are described below:

1. The DDS user writes data by invoking the write operation on the DDS DataWriter.
2. The DDS DataWriter invokes the new_change operation on the RTPS Writer to create a new CacheChange. Each CacheChange is identified uniquely by a SequenceNumber.
3. The new_change operation returns.
4. The DDS DataWriter uses the add_change operation to store the CacheChange into the RTPS Writer’s HistoryCache.
5. The add_change operation returns.
6. The write operation returns, the user has completed the action of writing Data.
7. The RTPS Writer sends the contents of the CacheChange changes to the RTPS Reader using the Data Submessage and requests an acknowledgment by also sending a Heartbeat Submessage.
8. The RTPS Reader receives the Data message and, assuming that the resource limits allow that, places the CacheChange into the reader’s HistoryCache using the add_change operation.
9. The add_change operation returns. The CacheChange is visible to the DDS DataReader and the DDS user. The conditions for this depend on the reliabilityLevel attribute of the RTPS Reader.
   a. For a RELIABLE DDS DataReader, changes in its RTPS Reader’s HistoryCache are made visible to the user application only when all previous changes (i.e., changes with smaller sequence numbers) are also visible.
   b. For a BEST_EFFORT DDS DataReader, changes in its RTPS Reader’s HistoryCache are made visible to the user only if no future changes have already been made visible (i.e., if
there are no changes in the RTPS Receiver’s **HistoryCache** with a higher sequence number).

10. The DDS user is notified by one of the mechanisms described in the DDS Specification (e.g., by means of a listener or a **WaitSet**) and initiates reading of the data by calling the **take** operation on the DDS **DataReader**.

11. The DDS **DataReader** accesses the change using the **get_change** operation on the **HistoryCache**.

12. The **get_change** operation returns the **CacheChange** to the DataReader.

13. The **take** operation returns the data to the DDS user.

14. The RTPS **Reader** sends an **AckNack** message indicating that the **CacheChange** was placed into the Reader’s **HistoryCache**. The **AckNack** message contains the **GUID** of the RTPS **Reader** and the **SequenceNumber** of the change. This action is independent from the notification to the DDS user and the reading of the data by the DDS user. It could have occurred before or concurrently with that.

15. The **StatefulWriter** records that the RTPS **Reader** has received the **CacheChange** and adds it to the set of **acked_changes** maintained by the **ReaderProxy** using the **acked_changes_set** operation.

16. The DDS user invokes the **return_loan** operation on the DataReader to indicate that it is no longer using the data it retrieved by means of the previous **take** operation. This action is independent from the actions on the writer side as it is initiated by the DDS user.

17. The DDS **DataReader** uses the **remove_change** operation to remove the data from the **HistoryCache**.

18. The **remove_change** operation returns.

19. The **return_loan** operation returns.

20. The DDS **DataWriter** uses the operation **is_acked_by_all** to determine which **CacheChanges** have been received by all the RTPS **Reader** endpoints matched with the **StatefulWriter**.

21. The **is_acked_by_all** returns and indicates that the change with the specified ‘seq_num’ **SequenceNumber** has been acknowledged by all RTPS **Reader** endpoints.

22. The DDS DataWriter uses the operation **remove_change** to remove the change associated with ‘seq_num’ from the RTPS Writer’s **HistoryCache**. In doing this, the DDS DataWriter also takes into account other DDS QoS such as **DURABILITY**.

23. The operation **remove_change** returns.

The description above did not model some of the interactions between the DDS DataReader and the RTPS **Reader**; for example, the mechanism used by the RTPS Reader to alert to the DataReader that it should call read or take to check whether new changes have been received (i.e., what causes step 10 to be taken).

Also unmodeled are some interactions between the DDS DataWriter and the RTPS **Writer**: such as the mechanism used by the RTPS **Writer** to alert to the DataWriter that it should check whether a particular change has been fully acknowledged such that it can be removed from the **HistoryCache** (i.e., what causes step 20 above to be initiated).

The aforementioned interactions are not modeled because they are internal to the implementation of the middleware and have no effect on the RTPS protocol.

### 8.4.2 Behavior Required for Interoperability

This sub clause describes the requirements that all implementations of the RTPS protocol must satisfy in order to be:

- compliant with the protocol specification
- interoperable with other implementations
The scope of these requirements is limited to message exchanges between RTPS implementations by different vendors. For message exchanges between implementations by the same vendor, vendors may opt for a non-compliant implementation or may use a proprietary protocol instead.

### 8.4.2.1 General Requirements

The following requirements apply to all RTPS Entities.

#### 8.4.2.1.1 All communications must take place using RTPS Messages

No other messages can be used than the RTPS Messages defined in 8.3. The required contents, validity and interpretation of each Message is defined by the RTPS specification.

Vendors may extend Messages for vendor specific needs using the extension mechanisms provided by the protocol (see 8.6). This does not affect interoperability.

#### 8.4.2.1.2 All implementations must implement the RTPS Message Receiver

Implementations must implement the rules followed by the RTPS Message Receiver, as introduced in 8.3.4, to interpret Submessages within the RTPS Message and maintain the state of the Message Receiver.

This requirement also includes proper Message formatting by preceding Entity Submessages with Interpreter Submessages when required for proper interpretation of the former, as defined in 8.3.8.

#### 8.4.2.1.3 The timing characteristics of all implementations must be tunable

Depending on the application requirements, deployment configuration and underlying transports, the end-user may want to tune the timing characteristics of the RTPS protocol.

Therefore, where the requirements on the protocol behavior allow delayed responses or specify periodic events, implementations must allow the end-user to tune those timing characteristics.

#### 8.4.2.1.4 Implementations must implement the Simple Participant and Endpoint Discovery Protocols

Implementations must implement the Simple Participant and Endpoint Discovery Protocols to enable the discovery of remote Endpoints (see 8.5).

RTPS allows the use of different Participant and Endpoint Discovery Protocols, depending on the deployment needs of the application. For the purpose of interoperability, implementations must implement at least the Simple Participant Discovery Protocol and Simple Endpoint Discovery Protocol (see 8.5.1).

### 8.4.2.2 Required RTPS Writer Behavior

The following requirements apply to RTPS Writers only. Unless indicated, the requirements apply to both reliable and best-effort Writers.

#### 8.4.2.2.1 Writers must not send data out-of-order

A Writer must send out data samples in the order they were added to its HistoryCache.

#### 8.4.2.2.2 Writers must include in-line QoS values if requested by a Reader

A Writer must honor a Reader’s request to receive data messages with in-line QoS.

#### 8.4.2.2.3 Writers must send periodic HEARTBEAT Messages (reliable only)

A Writer must periodically inform each matching reliable Reader of the availability of a data sample by sending a periodic HEARTBEAT Message that includes the sequence number of the available sample. If no samples are available, no HEARTBEAT Message needs to be sent.
For strict reliable communication, the **Writer** must continue to send HEARTBEAT Messages to a **Reader** until the **Reader** has either acknowledged receiving all available samples or has disappeared. In all other cases, the number of HEARTBEAT Messages sent can be implementation specific and may be finite.

8.4.2.4 Writers must eventually respond to a negative acknowledgment (reliable only)

When receiving an ACKNACK Message indicating a **Reader** is missing some data samples, the **Writer** must respond by either sending the missing data samples, sending a GAP message when the sample is not relevant, or sending a HEARTBEAT message when the sample is no longer available.

The **Writer** may respond immediately or choose to schedule the response for a certain time in the future. It can also coalesce related responses so there need not be a one-to-one correspondence between an ACKNACK Message and the **Writer**’s response. These decisions and the timing characteristics are implementation specific.

8.4.2.5 Sending Heartbeats and Gaps with Writer Group Information

A **Writer** belonging to a **Group** shall send HEARTBEAT or GAP Submessages to its matched **Readers** even if the **Reader** has acknowledged all of that **Writer**’s samples. This is necessary for the **Subscriber** to detect the group sequence numbers that are not available in that **Writer**. The exception to this rule is when the **Writer** has sent DATA or DATA_FRAG Submessages that contain the same information.

8.4.2.3 Required RTPS Reader Behavior

A best-effort **Reader** is completely passive as it only receives data and does not send messages itself. Therefore, the requirements below only apply to reliable **Readers**.

8.4.2.3.1 Readers must respond eventually after receiving a HEARTBEAT with final flag not set

Upon receiving a HEARTBEAT Message with final flag not set, the **Reader** must respond with an ACKNACK Message. The ACKNACK Message may acknowledge having received all the data samples or may indicate that some data samples are missing.

The response may be delayed to avoid message storms.

8.4.2.3.2 Readers must respond eventually after receiving a HEARTBEAT that indicates a sample is missing

Upon receiving a HEARTBEAT Message, a **Reader** that is missing some data samples must respond with an ACKNACK Message indicating which data samples are missing. This requirement only applies if the **Reader** can accommodate these missing samples in its cache and is independent of the setting of the final flag in the HEARTBEAT Message.

The response may be delayed to avoid message storms.

The response is not required when a liveliness HEARTBEAT has both liveliness and final flags set to indicate it is a liveliness-only message.

8.4.2.3.3 Once acknowledged, always acknowledged

Once a **Reader** has positively acknowledged receiving a sample using an ACKNACK Message, it can no longer negatively acknowledge that same sample at a later point.

Once a **Writer** has received positive acknowledgement from all **Readers**, the **Writer** can reclaim any associated resources. However, if a **Writer** receives a negative acknowledgement to a previously positively acknowledged sample, and the **Writer** can still service the request, the **Writer** should send the sample.

8.4.2.3.4 Readers can only send an ACKNACK Message in response to a HEARTBEAT Message

In steady state, an ACKNACK Message can only be sent as a response to a HEARTBEAT Message from a **Writer**. ACKNACK Messages can be sent from a **Reader** when it first discovers a **Writer** as an optimization. **Writers** are not required to respond to these pre-emptive ACKNACK Messages.
8.4.3 Implementing the RTPS Protocol

The RTPS specification states that a compliant implementation of the protocol need only satisfy the requirements presented in 8.4.2. Therefore, the behavior of actual implementations may differ as a function of the design trade-offs made by each implementation.

The Behavior Module of the RTPS specification defines two reference implementations:

Stateless Reference Implementation: The Stateless Reference Implementation is optimized for scalability. It keeps virtually no state on remote entities and therefore scales very well with large systems. This involves a trade-off, as improved scalability and reduced memory usage may require additional bandwidth usage. The Stateless Reference Implementation is ideally suited for best-effort communication over multicast.

Stateful Reference Implementation: The Stateful Reference Implementation maintains full state on remote entities. This approach minimizes bandwidth usage, but requires more memory and may imply reduced scalability. In contrast to the Stateless Reference Implementation, it can guarantee strict reliable communication and is able to apply QoS-based or content-based filtering on the Writer side.

Both reference implementations are described in detail in the sub clauses that follow.

Actual implementations need not necessarily follow the reference implementations. Depending on how much state is maintained, implementations may be a combination of the reference implementations.

For example, the Stateless Reference Implementation maintains minimal info and state on remote Entities. As such, it is not able to perform time-based filtering on the Writer side as this requires keeping track of each remote Reader and its properties. It is also not able to drop out-of-order samples on the Reader side as this requires keeping track of the largest sequence number received from each remote Writer. Some implementations may mimic the Stateless Reference Implementation, but choose to store enough additional state to be able to avoid some of the above limitations. The required additional information can be stored in a permanent fashion, in which case the implementation approaches the Stateful Reference Implementation, or can be slowly aged and kept around on an as needed basis to approximate, to the extent possible, the behavior that would result if the state were maintained.

Regardless of the actual implementation, in order to guarantee interoperability, it is important that all implementations, including both reference implementations, satisfy the requirements presented in 8.4.2.

8.4.4 The Behavior of a Writer with respect to each matched Reader

The behavior of an RTPS Writer with respect to each matched Reader depends on the setting of the reliabilityLevel attribute in the RTPS Writer and RTPS Reader. This controls whether a best-effort or a reliable protocol is used.

Not all possible combinations of the reliabilityLevel are possible. An RTPS Writer cannot be matched to an RTPS Reader unless either the RTPS Writer has the reliabilityLevel set to RELIABLE, or else both the RTPS Writer and RTPS Reader have the reliabilityLevel set to BEST_EFFORT. This is because the DDS specification states that a BEST_EFFORT DDS DataWriter can only be matched with a BEST_EFFORT DDS DataReader and a RELIABLE DDS DataWriter can be matched with both a RELIABLE and a BEST_EFFORT DDS DataReader.

As mentioned in 8.4.3, whether a Writer can be matched to a Reader does not depend on whether both use the same implementation of the RTPS protocol. That is, a Stateful Writer is able to communicate with a Stateless Reader and vice versa.

8.4.5 Notational Conventions

The reference implementations are described using UML sequence charts and state-diagrams. These diagrams use some abbreviations to refer to the RTPS Entities. The abbreviations used are listed in Table 8.51.
Table 8.51 - Abbreviations used in the sequence charts and state diagrams of the Behavior Module

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Meaning</th>
<th>Example usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>DW</td>
<td>DDS DataWriter</td>
<td>DW::write</td>
</tr>
<tr>
<td>DR</td>
<td>DDS DataReader</td>
<td>DR::read</td>
</tr>
<tr>
<td>W</td>
<td>RTPS Writer</td>
<td>W::heartbeatPeriod</td>
</tr>
<tr>
<td>RP</td>
<td>RTPS ReaderProxy</td>
<td>RP::unicastLocatorList</td>
</tr>
<tr>
<td>RL</td>
<td>RTPS ReaderLocator</td>
<td>RL::locator</td>
</tr>
<tr>
<td>R</td>
<td>RTPS Reader</td>
<td>R::heartbeatResponseDelay</td>
</tr>
<tr>
<td>WP</td>
<td>RTPS WriterProxy</td>
<td>WP::remoteWriterGuid</td>
</tr>
<tr>
<td>WHC</td>
<td>HistoryCache of RTPS Writer</td>
<td>WHC::changes</td>
</tr>
<tr>
<td>RHC</td>
<td>HistoryCache of RTPS Reader</td>
<td>RHC::changes</td>
</tr>
</tbody>
</table>

8.4.6 Type Definitions

The Behavior Module introduces the following additional types.

Table 8.52 - Types definitions for the Behavior Module

<table>
<thead>
<tr>
<th>Types used within the RTPS Model classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute type</td>
</tr>
<tr>
<td>Duration_t</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ChangeForReaderStatusKind</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ChangeFromWriterStatusKind</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>InstanceHandle_t</td>
</tr>
<tr>
<td>ParticipantMessageData</td>
</tr>
</tbody>
</table>

8.4.7 RTPS Writer Reference Implementations

The RTPS Writer Reference Implementations are based on specializations of the RTPS Writer class, first introduced in 8.2. This sub clause describes the RTPS Writer and all additional classes used to model the RTPS Writer Reference Implementations. The actual behavior is described in 8.4.8 and 8.4.9.
8.4.7.1 RTPS Writer

RTPS Writer specializes RTPS Endpoint and represents the actor that sends CacheChange messages to the matched RTPS Reader endpoints. The Reference Implementations StatelessWriter and StatefulWriter specialize RTPS Writer and differ in the knowledge they maintain about the matched Reader endpoints.

Figure 8.16 - RTPS Writer Endpoints

Table 8.53 describes the attributes of the RTPS Writer.
### RTPS Writer Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>meaning</th>
<th>relation to DDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushMode</td>
<td>bool</td>
<td>Configures the mode in which the Writer operates. If pushMode==true, then the Writer will push changes to the reader. If pushMode==false, changes will only be announced via heartbeats and only be sent as response to the request of a reader.</td>
<td>N/A (automatically configured)</td>
</tr>
<tr>
<td>heartbeatPeriod</td>
<td>Duration_t</td>
<td>Protocol tuning parameter that allows the RTPS Writer to repeatedly announce the availability of data by sending a Heartbeat Message.</td>
<td>N/A (automatically configured)</td>
</tr>
<tr>
<td>nackResponseDelay</td>
<td>Duration_t</td>
<td>Protocol tuning parameter that allows the RTPS Writer to delay the response to a request for data from a negative acknowledgment.</td>
<td>N/A (automatically configured)</td>
</tr>
<tr>
<td>nackSuppression</td>
<td>Duration_t</td>
<td>Protocol tuning parameter that allows the RTPS Writer to ignore requests for data from negative acknowledgments that arrive ‘too soon’ after the corresponding change is sent.</td>
<td>N/A (automatically configured)</td>
</tr>
<tr>
<td>lastChangeSequenceNumber</td>
<td>SequenceNumber_t</td>
<td>Internal counter used to assign increasing sequence number to each change made by the Writer.</td>
<td>N/A (used as part of the logic of the virtual machine)</td>
</tr>
<tr>
<td>writer_cache</td>
<td>HistoryCache</td>
<td>Contains the history of CacheChange changes for this Writer.</td>
<td>N/A</td>
</tr>
<tr>
<td>dataMaxSize</td>
<td>Serialized</td>
<td>Optional attribute that indicates the maximum size of any SerializedPayload that may be sent by the Writer.</td>
<td>N/A (automatically configured)</td>
</tr>
</tbody>
</table>

The attributes of the RTPS Writer allow for fine-tuning of the protocol behavior. The operations of the RTPS Writer are described in Table 8.54.
Table 8.54 - RTPS Writer operations

<table>
<thead>
<tr>
<th>operation name</th>
<th>parameter list</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>new</td>
<td>&lt;return value&gt;</td>
<td>Writer</td>
</tr>
<tr>
<td></td>
<td>attribute_values</td>
<td>Set of attribute values required by the Writer and all the super classes.</td>
</tr>
<tr>
<td>new_change</td>
<td>&lt;return value&gt;</td>
<td>CacheChange</td>
</tr>
<tr>
<td></td>
<td>kind</td>
<td>ChangeKind_t</td>
</tr>
<tr>
<td></td>
<td>data</td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td>inlineQos</td>
<td>ParameterList</td>
</tr>
<tr>
<td></td>
<td>handle</td>
<td>InstanceHandle_t</td>
</tr>
</tbody>
</table>

The following sub clauses provide details on the operations.

8.4.7.1.1 Default Timing-Related Values

The following timing-related values are used as the defaults in order to facilitate ‘out-of-the-box’ interoperability between implementations.

```plaintext
nackResponseDelay.sec = 0;
nackResponseDelay.nanosec = 200 * 1000 * 1000; //200 milliseconds
nackSuppressionDuration.sec = 0;
nackSuppressionDuration.nanosec = 0;
```

8.4.7.1.2 new

This operation creates a new RTPS Writer.

The newly-created writer ‘this’ is initialized as follows:

```plaintext
this.guid := <as specified in the constructor>;
this.unicastLocatorList := <as specified in the constructor>;
this.multicastLocatorList := <as specified in the constructor>;
this.reliabilityLevel := <as specified in the constructor>;
this.topicKind := <as specified in the constructor>;
this.pushMode := <as specified in the constructor>;
this.heartbeatPeriod := <as specified in the constructor>;
this.nackResponseDelay := <as specified in the constructor>;
this.nackSuppressionDuration := <as specified in the constructor>;
this.lastChangeSequenceNumber := 0;
this.writer_cache := new HistoryCache;
```

8.4.7.1.3 new_change

This operation creates a new CacheChange to be appended to the RTPS Writer’s HistoryCache. The sequence number of the CacheChange is automatically set to be the sequenceNumber of the previous change plus one.

This operation returns the new change.

This operation performs the following logical steps:

```plaintext
++this.lastChangeSequenceNumber;
    a_change := new CacheChange(kind, this.guid, this.lastChangeSequenceNumber,
                             data, inlineQos, handle);
RETURN a_change;
```
8.4.7.2 RTPS StatelessWriter

Specialization of RTPS Writer used for the Stateless Reference Implementation. The RTPS StatelessWriter has no knowledge of the number of matched readers, nor does it maintain any state for each matched RTPS Reader endpoint. The RTPS StatelessWriter maintains only the RTPS ReaderLocator list that should be used to send information to the matched readers.

Table 8.55 - RTPS StatelessWriter attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>meaning</th>
<th>relation to DDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>reader_locators</td>
<td>ReaderLocator[*]</td>
<td>The StatelessWriter maintains the list of locators to which it sends the CacheChanges. This list may include both unicast and multicast locators.</td>
<td>N/A (Automatically configured)</td>
</tr>
</tbody>
</table>

The RTPS StatelessWriter is useful for situations where (a) the writer’s HistoryCache is small, or (b) the communication is best-effort, or (c) the writer is communicating via multicast to a large number of readers.

The virtual machine interacts with the StatelessWriter using the operations in Table 8.56

Table 8.56 - StatelessWriter operations

<table>
<thead>
<tr>
<th>StatelessWriter operations</th>
<th>operation name</th>
<th>parameter list</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>new</td>
<td>&lt;return value&gt;</td>
<td>StatelessWriter</td>
<td></td>
</tr>
<tr>
<td>attribute_values</td>
<td>Set of attribute values required by the StatelessWriter and all the super classes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reader_locator_add</td>
<td>&lt;return value&gt;</td>
<td>void</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a_locator</td>
<td>Locator_t</td>
<td></td>
</tr>
<tr>
<td>reader_locator_remove</td>
<td>&lt;return value&gt;</td>
<td>void</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a_locator</td>
<td>Locator_t</td>
<td></td>
</tr>
<tr>
<td>unsent_changes_reset</td>
<td>&lt;return value&gt;</td>
<td>void</td>
<td></td>
</tr>
</tbody>
</table>

8.4.7.2.1 new

This operation creates a new RTPS StatelessWriter.

In addition to the initialization performed on the RTPS Writer super class (8.4.7.1.2), the newly-created StatelessWriter ‘this’ is initialized as follows:

```c
this.reader_locators := <empty>;
```

8.4.7.2.2 reader_locator_add

This operation adds the ReaderLocator a_locator to the StatelessWriter::reader_locators.

```c
ADD a_locator TO {this.reader_locators};
```
8.4.7.2.3 reader_locator_remove

This operation removes the ReaderLocator \( a\_locator \) from the StatelessWriter::reader Locators.

\[
\text{REMOVE } a\_locator \text{ FROM } \{\text{this.reader_locators}\};
\]

8.4.7.2.4 unsent_changes_reset

This operation resets the ‘highestSentChangeSN’ for all the ReaderLocators in the StatelessWriter::reader_locators. This operation is useful when called periodically to cause the StatelessWriter to keep re-sending all available changes in its HistoryCache.

\[
\text{FOREACH readerLocator in } \{\text{this.reader_locators}\} \text{ DO}
\]

\[
\text{readerLocator.highestSentChangeSN} := 0
\]

8.4.7.3 RTPS ReaderLocator

Valuetype used by the RTPS StatelessWriter to keep track of the locators of all matching remote Readers.

Table 8.57 - RTPS ReaderLocator attributes

<table>
<thead>
<tr>
<th>RTPS ReaderLocator</th>
<th>attribute</th>
<th>type</th>
<th>meaning</th>
<th>relation to DDS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>highestSentChangeSN</td>
<td>SequenceNumber_t</td>
<td>The highest sequence number of the changes that have been sent to the ReaderLocator</td>
<td>N/A Used to implement the behavior of the RTPS protocol.</td>
</tr>
<tr>
<td></td>
<td>requestedChanges</td>
<td>SequenceNumber_t[*</td>
<td>A list of sequence numbers representing changes that were requested by remote Readers at this ReaderLocator.</td>
<td>N/A Automatically configured</td>
</tr>
<tr>
<td></td>
<td>locator</td>
<td>Locator_t</td>
<td>Unicast or multicast locator through which the readers represented by this ReaderLocator can be reached.</td>
<td>N/A Automatically configured</td>
</tr>
<tr>
<td></td>
<td>expectsInlineQos</td>
<td>bool</td>
<td>Specifies whether the readers represented by this ReaderLocator expect inline QoS to be sent with every Data Message.</td>
<td></td>
</tr>
</tbody>
</table>

The virtual machine interacts with the ReaderLocator using the operations in Table 8.58.

Table 8.58 - RTPS ReaderLocator operations

<table>
<thead>
<tr>
<th>ReaderLocator operations</th>
<th>operation name</th>
<th>parameter list</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>new</td>
<td>&lt;return value&gt;</td>
<td>ReaderLocator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>attribute_values</td>
<td>Set of attribute values required by the ReaderLocator.</td>
</tr>
<tr>
<td></td>
<td>next_requested_change</td>
<td>&lt;return value&gt;</td>
<td>SequenceNumber_t</td>
</tr>
<tr>
<td></td>
<td>next_unsent_change</td>
<td>&lt;return value&gt;</td>
<td>SequenceNumber_t</td>
</tr>
<tr>
<td></td>
<td>requested_changes</td>
<td>&lt;return value&gt;</td>
<td>SequenceNumber_t[*</td>
</tr>
<tr>
<td>Function</td>
<td>Return Type</td>
<td>Parameters</td>
<td></td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------</td>
<td>---------------------</td>
<td></td>
</tr>
<tr>
<td>requested_changes_set</td>
<td>&lt;return value&gt;</td>
<td>void</td>
<td></td>
</tr>
<tr>
<td>req_seq_num_set</td>
<td>SequenceNumber_t[*]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsent_changes</td>
<td>&lt;return value&gt;</td>
<td>boolean</td>
<td></td>
</tr>
</tbody>
</table>

### 8.4.7.3.1 new

This operation creates a new RTPS ReaderLocator. The newly-created ReaderLocator ‘this’ is initialized as follows:

```plaintext
this.requested_changes := <empty>;  
this.highestSentChangeSN := SEQUENCE_NUMBER_INVALID;  
this.locator := <as specified in the constructor>;  
this.expectsInlineQos := <as specified in the constructor;  
```

### 8.4.7.3.2 next_requested_change

This operation returns the lowest sequence number of the requested_changes. This represents the next repair change that should be sent to the RTPS Reader located at this ReaderLocator in response to a previous AckNack message (see 8.3.8.1) from the Reader.

```plaintext
return MIN( this.requested_changes() )  
```

### 8.4.7.3.3 next_unsent_change

This operation returns the lowest sequence number of all the changes in the Writer HistoryCache that have a sequence number greater than the ReaderLocator ‘highestSentChangeSN’. This represents the next change that should be sent to the RTPS Reader located at this ReaderLocator.

```plaintext
unsent_changes :=  
{ changes SUCH_THAT change.sequenceNumber > this.highestSentChangeSN }  
IF unsent_changes == <empty>  return  SEQUENCE_NUMBER_INVALID  
ELSE return MIN { unsent_changes.sequenceNumber }  
```

### 8.4.7.3.4 requested_changes

This operation returns the list of sequence numbers for changes that were requested by the RTPS Readers at this ReaderLocator using an ACKNACK Message.

```plaintext
return this.requested_changes;  
```

### 8.4.7.3.5 requested_changes_set

This operation adds the set of change sequence numbers ‘req_seq_num_set’ to the requested_changes list.

```plaintext
FOR_EACH seq_num IN req_seq_num_set DO  
  ADD seq_num TO this.requested_changes;  
END  
```

### 8.4.7.3.6 unsent_changes

This operation returns TRUE if there are changes in the writer’s HistoryCache that have not been sent yet to this ReaderLocator, otherwise it returns FALSE.

```plaintext
return this.next_unsent_change() != SEQUENCE_NUMBER_INVALID;  
```

### 8.4.7.4 RTPS StatefulWriter

Specialization of RTPS Writer used for the Stateful Reference Implementation. The RTPS StatefulWriter is configured with the knowledge of all matched RTPS Reader endpoints and maintains state on each matched RTPS Reader endpoint.

By maintaining state on each matched RTPS Reader endpoint, the RTPS StatefulWriter can determine whether all matched RTPS Reader endpoints have received a particular CacheChange and can be optimal in its use of...
network bandwidth by avoiding to send announcements to readers that have received all the changes in the
writer’s HistoryCache. The information it maintains also simplifies QoS-based filtering on the Writer side. The
attributes specific to the StatefulWriter are described in Table 8.59.

Table 8.59 - RTPS StatefulWriter Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>meaning</th>
<th>relation to DDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>matched_readers</td>
<td>ReaderProxy[*]</td>
<td>The StatefulWriter keeps track of all the RTPS Readers matched with it. Each matched reader is represented by an instance of the ReaderProxy class.</td>
<td>N/A Automatically configured</td>
</tr>
</tbody>
</table>

The virtual machine interacts with the StatefulWriter using the operations in Table 8.60.

Table 8.60 - StatefulWriter Operations

<table>
<thead>
<tr>
<th>StatefulWriter operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation name</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>new</td>
</tr>
<tr>
<td>attribute_values</td>
</tr>
<tr>
<td>matched_reader_add</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>matched_reader_remove</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>matched_reader_lookup</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>is_acked_by_all</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

8.4.7.4.1 new

This operation creates a new RTPS StatefulWriter. In addition to the initialization performed on the RTPS Writer super class (8.4.7.1.2), the newly-created StatefulWriter ‘this’ is initialized as follows:

```
this.matched_readers := <empty>;
```

8.4.7.4.2 is_acked_by_all

This operation takes a SequenceNumber_t a_change_seq_num as a parameter and determines whether all the ReaderProxy have acknowledged the CacheChange with that sequence number. The operation will return true if all ReaderProxy have acknowledged the corresponding CacheChange and false otherwise.

```
return true IF and only IF
FOREACH proxy IN this.matched_readers
   a_change_seq_num IN proxy.acknowledged_changes
```

8.4.7.4.3 matched_reader_add

This operation adds the ReaderProxy a_reader_proxy to the set StatefulWriter::matched_readers.
ADD a_reader_proxy TO {this.matched_readers};

8.4.7.4.4 matched_reader_remove

This operation removes the ReaderProxy a_reader_proxy from the set StatefulWriter::matched_readers.
    REMOVE a_reader_proxy FROM {this.matched_readers};
    delete proxy;

8.4.7.4.5 matched_reader_lookup

This operation finds the ReaderProxy with GUID_t a_reader_guid from the set StatefulWriter::matched_readers.
    FIND proxy IN this.matched_readers
      SUCH-THAT (proxy.remoteReaderGuid == a_reader_guid);
    return proxy;

8.4.7.5 RTPS ReaderProxy

The RTPS ReaderProxy class represents the information an RTPS StatefulWriter maintains on each matched RTPS Reader. The attributes of the RTPS ReaderProxy are described in Table 8.61.

Table 8.61 - RTPS ReaderProxy Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>meaning</th>
<th>relation to DDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>remoteReaderGuid</td>
<td>GUID_t</td>
<td>Identifies the remote matched RTPS Reader that is represented by the ReaderProxy.</td>
<td>N/A Configured by discovery</td>
</tr>
<tr>
<td>remoteGroupEntityId</td>
<td>EntityId_t</td>
<td>Identifies the group to which the matched Reader belongs</td>
<td>The EntityId of the Subscriber to which this DataReader belongs.</td>
</tr>
<tr>
<td>unicastLocatorList</td>
<td>Locator_t[*]</td>
<td>List of unicast locators (transport, address, port combinations) that can be used to send messages to the matched RTPS Reader. The list may be empty.</td>
<td>N/A Configured by discovery</td>
</tr>
<tr>
<td>multicastLocatorList</td>
<td>Locator_t[*]</td>
<td>List of multicast locators (transport, address, port combinations) that can be used to send messages to the matched RTPS Reader. The list may be empty.</td>
<td>N/A Configured by discovery</td>
</tr>
<tr>
<td>highestSentChangeSN</td>
<td>SequenceNumber_t</td>
<td>The highest sequence number of the changes that have been sent to the matched RTPS Reader.</td>
<td>N/A Used to implement the behavior of the RTPS protocol.</td>
</tr>
<tr>
<td>requestedChanges</td>
<td>SequenceNumber_t[*]</td>
<td>A list of sequence numbers representing changes that were requested by the matched RTPS Reader.</td>
<td>N/A Used to implement the behavior of the RTPS protocol.</td>
</tr>
<tr>
<td>acknowledgedChanges</td>
<td>SequenceNumber_t[*]</td>
<td>A list of sequence numbers representing changes that have been acknowledged by the matched RTPS Reader.</td>
<td>N/A</td>
</tr>
</tbody>
</table>
The matching of an RTPS StatefulWriter with an RTPS Reader means that the RTPS StatefulWriter will send the CacheChange changes in the writer’s HistoryCache to the matched RTPS Reader represented by the ReaderProxy. The matching is a consequence of the match of the corresponding DDS entities. That is, the DDS DataWriter matches a DDS DataReader by Topic, has compatible QoS, and is not being explicitly ignored by the application that uses DDS.

The virtual machine interacts with the ReaderProxy using the operations in Table 8.62.

Table 8.62 - ReaderProxy Operations

<table>
<thead>
<tr>
<th>operation name</th>
<th>parameter list</th>
<th>parameter type</th>
</tr>
</thead>
<tbody>
<tr>
<td>new</td>
<td>&lt;return value&gt;</td>
<td>ReaderProxy</td>
</tr>
<tr>
<td></td>
<td>attribute_values</td>
<td>Set of attribute values required by the ReaderProxy.</td>
</tr>
<tr>
<td>acked_changes_set</td>
<td>&lt;return value&gt;</td>
<td>void</td>
</tr>
<tr>
<td></td>
<td>committed_seq_num</td>
<td>SequenceNumber_t</td>
</tr>
<tr>
<td>next_requested_change</td>
<td>&lt;return value&gt;</td>
<td>SequenceNumber_t</td>
</tr>
<tr>
<td>next_unsent_change</td>
<td>&lt;return value&gt;</td>
<td>SequenceNumber_t</td>
</tr>
<tr>
<td>unsent_changes</td>
<td>&lt;return value&gt;</td>
<td>boolean</td>
</tr>
<tr>
<td>requested_changes</td>
<td>&lt;return value&gt;</td>
<td>SequenceNumber_t[*]</td>
</tr>
<tr>
<td>requested_changes_set</td>
<td>&lt;return value&gt;</td>
<td>void</td>
</tr>
<tr>
<td></td>
<td>req_seq_num_set</td>
<td>SequenceNumber_t[*]</td>
</tr>
<tr>
<td>unacked_changes</td>
<td>&lt;return value&gt;</td>
<td>boolean</td>
</tr>
</tbody>
</table>

8.4.7.5.1 new

This operation creates a new RTPS ReaderProxy. The newly-created reader proxy ‘this’ is initialized as follows:

```java
this.attributes := <as specified in the constructor>;
this.requested_changes := <empty>;
this.acknowledged_changes := <empty>;
this.highest_sent_seq_num := 0;
```
8.4.7.5.2  **acked_changes_set**

This operation modifies the ‘acknowledged_changes’ attribute to include all changes with sequence number smaller than or equal to the value ‘committed_seq_num’.

```
FOR EACH seq_num <= committed_seq_num DO
  ADD seq_num TO this.acknowledged_changes
```

8.4.7.5.3  **next_requested_change**

This operation returns the lowest sequence number in the ‘requested_changes’ attribute. This represents the next repair change that should be sent to the RTPS Reader represented by the ReaderProxy in response to a previous AckNack message (see 8.3.8.1) from the Reader.

```
return MIN( this.requested_changes() );
```

8.4.7.5.4  **next_unsent_change**

This operation returns the lowest sequence number of all the changes in the Writer HistoryCache that have a sequence number greater than the ReaderProxy ‘highest_sent_seq_num’. This represents the next change that should be sent to the RTPS Reader represented by the ReaderProxy.

```
unsent_changes :=
{ changes SUCH-THERE change.sequenceNumber > this.highest_sent_seq_num }

IF unsent_changes == <empty> return SEQUENCE_NUMBER_INVALID
ELSE return MIN { unsent_changes.sequenceNumber }
```

8.4.7.5.5  **requested_changes**

This operation returns the list of sequence numbers for changes that were requested by the RTPS Reader represented by the ReaderProxy using an ACKNACK Message.

```
return this.requested_changes
```

8.4.7.5.6  **requested_changes_set**

This operation modifies the ‘requested_changes’ attribute to include the set of changes with sequence numbers that appear in the parameter ‘req_seq_num_set’.

```
FOR EACH seq_num IN req_seq_num_set DO
  ADD seq_num TO this.requested_changes;
END
```

8.4.7.5.7  **unsent_changes**

This operation returns ‘true’ if there are changes in the writer’s HistoryCache that have not been sent yet to this ReaderProxy, otherwise it returns FALSE.

```
return ( this.next_unsent_change() != SEQUENCE_NUMBER_INVALID )
```

8.4.7.5.8  **unacked_changes**

This operation returns ‘true’ if there are changes in the writer’s HistoryCache that have not been acknowledged yet by the RTPS Reader represented by the ReaderProxy.

```
highest_available_seq_num := MAX { change.sequenceNumber }
highest_acked_seq_num := MAX { this.acknowledged_changes }

return ( highest_available_seq_num > highest_acked_seq_num )
```
8.4.8 RTPS StatelessWriter Behavior

8.4.8.1 Best-Effort StatelessWriter Behavior

The behavior of the Best-Effort RTPS StatelessWriter with respect to each ReaderLocator is described in Figure 8.17.

![State Machine Diagram](image)

**Figure 8.17 - Behavior of the Best-Effort StatelessWriter with respect to each ReaderLocator**

The state-machine transitions are listed in Table 8.63.

**Table 8.63 - Transitions for Best-effort StatelessWriter behavior with respect to each ReaderLocator**

<table>
<thead>
<tr>
<th>Transition</th>
<th>state</th>
<th>event</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>initial</td>
<td>RTPS Writer is configured with a ReaderLocator</td>
<td>idle</td>
</tr>
<tr>
<td>T2</td>
<td>idle</td>
<td>GuardCondition: RL::unsent_changes() == true</td>
<td>pushing</td>
</tr>
<tr>
<td>T3</td>
<td>pushing</td>
<td>GuardCondition: RL::unsent_changes() == false</td>
<td>idle</td>
</tr>
<tr>
<td>T4</td>
<td>pushing</td>
<td>GuardCondition: RL::can_send() == true</td>
<td>pushing</td>
</tr>
<tr>
<td>T5</td>
<td>any state</td>
<td>RTPS Writer is configured to no longer have the ReaderLocator</td>
<td>final</td>
</tr>
</tbody>
</table>

8.4.8.1.1 Transition T1

This transition is triggered by the configuration of an RTPS Best-Effort StatelessWriter ‘the_rtps_writer’ with an RTPS ReaderLocator. This configuration is done by the Discovery protocol (8.5) as a consequence of the discovery of a DDS DataReader that matches the DDS DataWriter that is related to ‘the_rtps_writer.’
The discovery protocol supplies the values for the ReaderLocator constructor parameters. The transition performs the following logical actions in the virtual machine:

```java
a_locator := new ReaderLocator( locator, expectsInlineQos );
the_rtps_writer.reader_locator_add( a_locator );
```

### 8.4.8.1.2 Transition T2

This transition is triggered by the guard condition \[\text{RL::unsent_changes() == true}\] indicating that there are some changes in the RTPS Writer HistoryCache that have not been sent to the RTPS ReaderLocator.

The transition performs no logical actions in the virtual machine.

### 8.4.8.1.3 Transition T3

This transition is triggered by the guard condition \[\text{RL::unsent_changes() == false}\] indicating that all changes in the RTPS Writer HistoryCache have been sent to the RTPS ReaderLocator. Note that this does not indicate that the changes have been received, only that an attempt was made to send them.

The transition performs no logical actions in the virtual machine.

### 8.4.8.1.4 Transition T4

This transition is triggered by the guard condition \[\text{RL::can_send() == true}\] indicating that the RTPS Writer ‘the_writer’ has the resources needed to send a change to the RTPS ReaderLocator ‘the_reader_locator.’

The transition performs the following logical actions in the virtual machine:

```java
a_change_seq_num := the_reader_locator.next_unsent_change();
IF ( a_change_seq_num > the_reader_locator.higuest_sent_seq_num +1 ) {
  GAP = new GAP( the_reader_locator.higuest_sent_seq_num +1,
                  a_change_seq_num - 1);
  GAP.readerId := ENTITYID_UNKNOWN;
  GAP.filteredCount := 0;
  sendto the_reader_locator.locator, GAP;
}

a_change := the_writer.writer_cache.get_change(a_change_seq_num);
DATA = new DATA(a_change);
IF (the_reader_locator.expectsInlineQos) {
  DATA.inlineQos := the_writer.related_dds_writer.qos;
  DATA.inlineQos += a_change.inlineQos;
}
DATA.readerId := ENTITYID_UNKNOWN;
sendto the_reader_locator.locator, DATA;
the_reader_locator.higuest_sent_seq_num := a_change_seq_num;
```

The next unsent change ‘a_change’ present in the Writer Cache may not not have a sequence number that matches the ReaderLocator (higuest_sent_seq_num + 1). This may happen when a CacheChanges is removed from the Writer cache. For example, when using HISTORY QoS set to KEEP_LAST with depth == 1, a new change will cause the DDS DataWriter to remove the previous change from the HistoryCache. In this case a GAP message is sent to indicate a range of sequence numbers not available to the Reader.

Since the GAP messages represent CacheChanges that are not present in the Writer cache, these changes do not appear counted in the GAP message filteredCount.

After the transition, the following post-conditions hold:

\[\text{the_reader_locator.higuest_sent_seq_num == a_change_seq_num}\]

### 8.4.8.1.5 Transition T5

This transition is triggered by the configuration of an RTPS Writer ‘the_rtps_writer’ to no longer send to the RTPS ReaderLocator ‘the_reader_locator.’ This configuration is done by the Discovery protocol (8.5) as a...
consequence of breaking a pre-existing match of a DDS DataReader with the DDS DataWriter related to 'the_rtps_writer.'

The transition performs the following logical actions in the virtual machine:

- `the_rtps_writer.reader_locator_remove(the_reader_locator);`
- `delete the_reader_locator;`

### 8.4.8.2 Reliable StatelessWriter Behavior

The behavior of the reliable RTPS `StatelessWriter` with respect to each `ReaderLocator` is described in Figure 8.18.

![Behavior of the Reliable StatelessWriter with respect to each ReaderLocator](image)

Figure 8.18 - Behavior of the Reliable StatelessWriter with respect to each ReaderLocator

The state-machine transitions are listed in Table 8.64.
Table 8.64 - Transitions for the Reliable StatelessWriter behavior with respect to each Reader Locator

<table>
<thead>
<tr>
<th>Transition</th>
<th>state</th>
<th>event</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>initial</td>
<td>RTPS Writer is configured with a ReaderLocator</td>
<td>announcing</td>
</tr>
<tr>
<td>T2</td>
<td>announcing</td>
<td>GuardCondition:</td>
<td>pushing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RL::unsent_changes() == true</td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>pushing</td>
<td>GuardCondition:</td>
<td>announcing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RL::unsent_changes() == false</td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td>pushing</td>
<td>GuardCondition:</td>
<td>pushing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RL::can_send() == true</td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td>announcing</td>
<td>after(W::heartbeatPeriod)</td>
<td>announcing</td>
</tr>
<tr>
<td>T6</td>
<td>waiting</td>
<td>ACKNACK message is received</td>
<td>waiting</td>
</tr>
<tr>
<td>T7</td>
<td>waiting</td>
<td>GuardCondition:</td>
<td>must_repair</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RL::requested_changes() != &lt;empty&gt;</td>
<td></td>
</tr>
<tr>
<td>T8</td>
<td>must_repair</td>
<td>ACKNACK message is received</td>
<td>must_repair</td>
</tr>
<tr>
<td>T9</td>
<td>must_repair</td>
<td>after(W::nackResponseDelay)</td>
<td>repairing</td>
</tr>
<tr>
<td>T10</td>
<td>repairing</td>
<td>GuardCondition:</td>
<td>repairing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RL::can_send() == true</td>
<td></td>
</tr>
<tr>
<td>T11</td>
<td>repairing</td>
<td>GuardCondition:</td>
<td>waiting</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RL::requested_changes() == &lt;empty&gt;</td>
<td></td>
</tr>
<tr>
<td>T12</td>
<td>any state</td>
<td>RTPS Writer is configured to no longer have the ReaderLocator</td>
<td>final</td>
</tr>
</tbody>
</table>

8.4.8.2.1 Transition T1

This transition is triggered by the configuration of an RTPS Reliable StatelessWriter ‘the_rtps_writer’ with an RTPS ReaderLocator. This configuration is done by the Discovery protocol (8.5, ‘Discovery Module’) as a consequence of the discovery of a DDS DataReader that matches the DDS DataWriter that is related to ‘the_rtps_writer.’

The discovery protocol supplies the values for the ReaderLocator constructor parameters. The transition performs the following logical actions in the virtual machine:

```plaintext
a_locator := new ReaderLocator( locator, expectsInlineQos );
the_rtps_writer.reader_locator_add( a_locator );
```

8.4.8.2.2 Transition T2

This transition is triggered by the guard condition [RL::unsent_changes() == true] indicating that there are some changes in the RTPS Writer HistoryCache that have not been sent to the ReaderLocator. The transition performs no logical actions in the virtual machine.

8.4.8.2.3 Transition T3

This transition is triggered by the guard condition [RL::unsent_changes == false] indicating that all changes in the RTPS Writer HistoryCache have been sent to the ReaderLocator. Note that this does not indicate that the changes have been received, only that there has been an attempt made to send them. The transition performs no logical actions in the virtual machine.
8.4.8.2.4 Transition T4

This transition is triggered by the guard condition [RL::can_send() == true] indicating that the RTPS Writer ‘the_writer’ has the resources needed to send a change to the RTPS ReaderLocator ‘the_reader_locator.’

The transition performs the following logical actions in the virtual machine:

```
a_change_seq_num := the_reader_locator.next_unsent_change();
IF ( a_change_seq_num > the_reader_locator.higuest_sent_seq_num + 1 ) {
  GAP = new GAP(the_reader_locator.higuest_sent_seq_num + 1,
                a_change_seq_num - 1);
  GAP.readerId := ENTITYID_UNKNOWN;
  GAP.filteredCount := 0;
  sendto the_reader_locator.locator, GAP;
}
a_change := the_writer.writer_cache.get_change(a_change_seq_num);
DATA = new DATA(a_change);
IF (the_reader_locator.expectsInlineQos) {
  DATA.inlineQos :=
    the_writer.related_dds_writer.qos;
}
DATA.readerId := ENTITYID_UNKNOWN;
sendto the_reader_locator.locator, DATA;
the_reader_locator.higuest_sent_seq_num := a_change_seq_num;
```

The next unsent change ‘a_change’ present in the Writer Cache may not have a sequence number that matches the ReaderLocator (higuest_seq_num + 1). This may happen when a CacheChanges is removed from the Writer cache. For example, when using HISTORY QoS set to KEEP_LAST with depth == 1, a new change will cause the DDS DataWriter to remove the previous change from the HistoryCache. In this case a GAP message is sent to indicate a range of sequence numbers not available to the Reader.

Since the GAP messages represent CacheChanges that are not present in the Writer cache, these changes do not appear counted in the GAP message filteredCount.

After the transition the following post-conditions hold:

```
the_reader_locator.higuest_sent_seq_num == a_change_seq_num
```

8.4.8.2.5 Transition T5

This transition is triggered by the firing of a periodic timer configured to fire each W::heartbeatPeriod.

The transition performs the following logical actions in the virtual machine for the Writer ‘the_rtps_writer’ and ReaderLocator ‘the_reader_locator.’

```
seq_num_min := the_rtps_writer.writer_cache.get_seq_num_min();
seq_num_max := the_rtps_writer.writer_cache.get_seq_num_max();
HEARTBEAT := new HEARTBEAT(the_rtps_writer.writerGuid, seq_num_min,
                           seq_num_max);
HEARTBEAT.FinalFlag := SET;
HEARTBEAT.readerId := ENTITYID_UNKNOWN;
sendto the_reader_locator, HEARTBEAT;
```

8.4.8.2.6 Transition T6

This transition is triggered by the reception of an ACKNACK message destined to the RTPS StatelessWriter ‘the_rtps_writer’ originating from some RTPS Reader.

The transition performs the following logical actions in the virtual machine:

```
FOREACH reply_locator_t IN { Receiver.unicastReplyLocatorList,
                            Receiver.multicastReplyLocatorList }
```
reader_locator := the_rtps_writer.reader_locator_lookup(reply_locator_t);
reader_locator.requested_changes_set(ACKNACK.readerSNState.set);

Note that the processing of this message uses the reply locators in the RTPS Receiver. This is the only source of information for the StatelessWriter to determine where to send the reply to. Proper functioning of the protocol requires that the RTPS Reader inserts an InfoReply Submessage ahead of the AckNack such that these fields are properly set.

8.4.8.2.7 Transition T7
This transition is triggered by the guard condition [RL::requested_changes() != <empty>] indicating that there are changes that have been requested by some RTPS Reader reachable at the RTPS ReaderLocator. The transition performs no logical actions in the virtual machine.

8.4.8.2.8 Transition T8
This transition is triggered by the reception of an ACKNACK message destined to the RTPS StatelessWriter ‘the_rtps_writer’ originating from some RTPS Reader. The transition performs the same logical actions performed by Transition T6 (8.4.8.2.6).

8.4.8.2.9 Transition T9
This transition is triggered by the firing of a timer indicating that the duration of W::nackResponseDelay has elapsed since the state must_repair was entered. The transition performs no logical actions in the virtual machine.

8.4.8.2.10 Transition T10
This transition is triggered by the guard condition [RL::can_send() == true] indicating that the RTPS Writer ‘the_writer’ has the resources needed to send a change to the RTPS ReaderLocator ‘the_reader_locator.’ The transition performs the following logical actions in the virtual machine.

```plaintext
a_change_seq_num := the_reader_locator.next_requested_change();
a_change := the_writer.writer_cache.get_change(a_change_seq_num);
IF ( a_change != <nil> ) {
    DATA = new DATA(a_change);
    IF (the_reader_locator.expectsInlineQos) {
        DATA.inlineQos := the_writer.related_dds_writer.qos;
        DATA.inlineQos += a_change.inlineQos;
    }
    DATA.readerId := ENTITYID_UNKNOWN;
    send to the_reader_locator.locator, DATA;
}
ELSE {
    GAP = new GAP(a_change.sequenceNumber);
    GAP.readerId := ENTITYID_UNKNOWN;
    GAP.filteredCount := 0;
    send to the_reader_locator.locator, GAP;
}
```

After the transition the following post-conditions hold:

```plaintext
( a_change_seq_num BELONGS_TO the_reader_locator.requested_changes() ) == FALSE
```

Note that it is possible that the requested change had already been removed from the HistoryCache by the DDS DataWriter. In that case, the StatelessWriter sends a GAP Message. The GAP message does not count the change in its message filteredCount because it corresponds to a change not present in the HistoryCache.
8.4.8.2.11 Transition T11

This transition is triggered by the guard condition \([\text{RL}::\text{requested\_changes}() == \langle\text{empty}\rangle]\) indicating that there are no further changes requested by an RTPS Reader reachable at the RTPS ReaderLocator. The transition performs no logical actions in the virtual machine.

8.4.8.2.12 Transition T12

This transition is triggered by the configuration of an RTPS Writer ‘the_rtps_writer’ to no longer send to the RTPS ReaderLocator ‘the_reader_locator.’ This configuration is done by the Discovery protocol (8.5) as a consequence of breaking a pre-existing match of a DDS DataReader with the DDS DataWriter related to ‘the_rtps_writer.’

The transition performs the following logical actions in the virtual machine:

```plaintext
the_rtps_writer.reader_locator_remove(the_reader_locator);
delete the_reader_locator;
```

8.4.9 RTPS StatefulWriter Behavior

8.4.9.1 Best-Effort StatefulWriter Behavior

The behavior of the Best-Effort RTPS StatefulWriter with respect to each matched RTPS Reader is described in Figure 8.19.

![Diagram of state machine transitions](image)

Figure 8.19 - Behavior of Best-Effort StatefulWriter with respect to each matched Reader

The state-machine transitions are listed in Table 8.65.
Table 8.65 - Transitions for Best-effort Stateful Writer behavior with respect to each matched Reader

<table>
<thead>
<tr>
<th>Transition</th>
<th>state</th>
<th>event</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>initial</td>
<td>RTPS Writer is configured with a matched RTPS Reader</td>
<td>idle</td>
</tr>
<tr>
<td>T2</td>
<td>idle</td>
<td>GuardCondition:</td>
<td>pushing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RP::unsent_changes() == true</td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>pushing</td>
<td>GuardCondition:</td>
<td>idle</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RP::unsent_changes() == false</td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td>pushing</td>
<td>GuardCondition:</td>
<td>pushing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RP::can_send() == true</td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td>ready</td>
<td>A new change was added to the RTPS Writer’s HistoryCache.</td>
<td>ready</td>
</tr>
<tr>
<td>T6</td>
<td>any state</td>
<td>RTPS Writer is configured to no longer be matched with the RTPS Reader</td>
<td>final</td>
</tr>
</tbody>
</table>

8.4.9.1.1 Transition T1

This transition is triggered by the configuration of an RTPS Writer ‘the_rtps_writer’ with a matching RTPS Reader. This configuration is done by the Discovery protocol (8.5) as a consequence of the discovery of a DDS DataReader that matches the DDS DataWriter that is related to ‘the_rtps_writer.’

The discovery protocol supplies the values for the ReaderProxy constructor parameters. The transition performs the following logical actions in the virtual machine:

```java
a_reader_proxy := new ReaderProxy( remoteReaderGuid, remoteGroupEntityId, expectsInlineQos, unicastLocatorList, multicastLocatorList);
the_rtps_writer.matched_reader_add(a_reader_proxy);
```

The ReaderProxy ‘a_reader_proxy’ is initialized as discussed in 8.4.7.5.

8.4.9.1.2 Transition T2

This transition is triggered by the guard condition [RP::unsent_changes() == true] indicating that there are some changes in the RTPS Writer HistoryCache that have not been sent to the RTPS Reader represented by the ReaderProxy.

Note that for a Best-Effort Writer, W::pushMode == true, as there are no acknowledgements. Therefore, the Writer always pushes out data as it becomes available.

The transition performs no logical actions in the virtual machine.

8.4.9.1.3 Transition T3

This transition is triggered by the guard condition [RP::unsent_changes() == false] indicating that all changes in the RTPS Writer HistoryCache have been sent to the RTPS Reader represented by the ReaderProxy. Note that this does not indicate that the changes have been received, only that there has been an attempt made to send them.

The transition performs no logical actions in the virtual machine.

8.4.9.1.4 Transition T4

This transition is triggered by the guard condition [RP::can_send() == true] indicating that the RTPS Writer ‘the_rtps_writer’ has the resources needed to send a change to the RTPS Reader represented by the ReaderProxy ‘the_reader_proxy.’
The transition performs the following logical actions in the virtual machine:

\[ a\text{\_change\_seq\_num} := \text{the\_reader\_proxy\_next\_unsent\_change}(); \]

\[ \text{if} \ (a\text{\_change\_seq\_num} > \text{the\_reader\_proxy\_higuest\_sent\_seq\_num} + 1) \ {\text{\{}} \]
\[ \text{GAP} = \text{new GAP}(\text{the\_reader\_locator\_higuest\_sent\_seq\_num} + 1, \]
\[ a\text{\_change\_seq\_num} - 1); \]
\[ \text{GAP\_readerId} := \text{ENTITYID\_UNKNOWN}; \]
\[ \text{GAP\_filteredCount} := 0; \]
\[ \text{send GAP}; \]
\[ \text{\}}} \]
\[ a\text{\_change} := \text{the\_writer\_writer\_cache\_get\_change}(a\text{\_change\_seq\_num}); \]

\[ \text{if} \ (\text{DDS\_FILTER}(\text{the\_reader\_proxy}, a\text{\_change}) \ {\text{\{}} \]
\[ \text{DATA} = \text{new DATA}(a\text{\_change}); \]
\[ \text{IF} \ (\text{the\_reader\_proxy\_expectsInlineQos}) \ {\text{\{}} \]
\[ \text{DATA\_inlineQos} := \text{the\_rtps\_writer\_related\_dds\_writer\_qos}; \]
\[ \text{DATA\_inlineQos} += a\text{\_change\_inlineQos}; \]
\[ \text{DATA\_readerId} := \text{ENTITYID\_UNKNOWN}; \]
\[ \text{send DATA}; \]
\[ \text{\}}} \]
\[ \text{else} \ {\text{\{}} \]
\[ \text{GAP} = \text{new GAP}(a\text{\_change\_sequenceNumber}); \]
\[ \text{GAP\_readerId} := \text{ENTITYID\_UNKNOWN}; \]
\[ \text{GAP\_filteredCount} := 1; \]
\[ \text{send GAP}; \]
\[ \text{\}}} \]
\[ \text{the\_reader\_proxy\_higuest\_sent\_seq\_num} := a\text{\_change\_seq\_num}; \]

The next unsent change ‘a_change’ present in the Writer Cache may not not have a sequence number that matches the ReaderProxy (higuest_sent_seq_num + 1). This may happen when a CacheChanges is removed from the Writer cache. For example, when using HISTORY QoS set to KEEP_LAST with depth == 1 and a new change for the same instance (key) replaces the previous one from the HistoryCache. In this case a GAP message is sent to indicate a range of sequence numbers not available to the Reader. This GAP represents CacheChanges that are not present in the Writer cache. Therefore, these changes do not appear counted in the GAP message filteredCount.

The next unsent change ‘a_change’ present in the Writer Cache may not pass a DDS_FILTER that indicates that the change is not relevant to the Reader. This may happen, for example, of the Reader specified a Content Filter. In this case a GAP is sent. However, this GAP represents CacheChanges that are present in the Writer cache but are not sent due to not being relevant. These changes do appear counted in the GAP message filteredCount.

The above logic is not meant to imply that each DATA Submessage is sent in a separate RTPS Message. Rather multiple Submessages can be combined into a single RTPS message.

After the transition, the following post-conditions hold:

\[ \text{the\_reader\_proxy\_higuest\_sent\_seq\_num} == a\text{\_change\_seq\_num} \]

8.4.9.1.5 Transition T5

This transition is triggered by the addition of a new CacheChange ‘a_change’ to the HistoryCache of the RTPS Writer ‘the_rtps_writer’ by the corresponding DDS DataWriter.

The transition performs the following logical actions in the virtual machine:

\[ \text{ADD a\_change TO the\_rtps\_writer\_writer\_cache}; \]

After the transition the following post-condition holds:

\[ \text{FOREACH proxy IN the\_rtps\_writer\_matched\_readers} \]
proxy.unsent_changes() == true)

8.4.9.1.6 Transition T6

This transition is triggered by the configuration of an RTPS Writer ‘the_rtps_writer’ to no longer be matched with the RTPS Reader represented by the ReaderProxy ‘the_reader_proxy’. This configuration is done by the Discovery protocol (8.5) as a consequence of breaking a pre-existing match of a DDS DataReader with the DDS DataWriter related to ‘the_rtps_writer.’

The transition performs the following logical actions in the virtual machine:

```c
the_rtps_writer.matched_reader_remove(the_reader_proxy);
delete the_reader_proxy;
```

8.4.9.2 Reliable StatefulWriter Behavior

The behavior of the Reliable RTPS StatefulWriter with respect to each matched RTPS Reader is described in Figure 8.20.
The state-machine transitions are listed in Table 8.66.

Table 8.66 - Transitions for Reliable StatefulWriter behavior with respect to each matched Reader

<table>
<thead>
<tr>
<th>Transition</th>
<th>state</th>
<th>event</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>initial</td>
<td>RTPS Writer is configured with a matched RTPS Reader</td>
<td>announcing</td>
</tr>
<tr>
<td>Transition (T)</td>
<td>State</td>
<td>Guard Condition</td>
<td>Action</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>T2</td>
<td>announcing</td>
<td>GuardCondition: RP::unsent_changes() == true</td>
<td>pushing</td>
</tr>
<tr>
<td>T3</td>
<td>pushing</td>
<td>GuardCondition: RP::unsent_changes() == false</td>
<td>announcing</td>
</tr>
<tr>
<td>T4</td>
<td>pushing</td>
<td>GuardCondition: RP::can_send() == true</td>
<td>pushing</td>
</tr>
<tr>
<td>T5</td>
<td>announcing</td>
<td>GuardCondition: RP::unacked_changes() == false</td>
<td>idle</td>
</tr>
<tr>
<td>T6</td>
<td>idle</td>
<td>GuardCondition: RP::unacked_changes() == true</td>
<td>announcing</td>
</tr>
<tr>
<td>T7</td>
<td>announcing</td>
<td>after(W::heartbeatPeriod)</td>
<td>announcing</td>
</tr>
<tr>
<td>T8</td>
<td>waiting</td>
<td>ACKNACK message is received</td>
<td>waiting</td>
</tr>
<tr>
<td>T9</td>
<td>waiting</td>
<td>GuardCondition: RP::requested_changes() != &lt;empty&gt;</td>
<td>must_repair</td>
</tr>
<tr>
<td>T10</td>
<td>must_repair</td>
<td>ACKNACK message is received</td>
<td>must_repair</td>
</tr>
<tr>
<td>T11</td>
<td>must_repair</td>
<td>after(W::nackResponseDelay)</td>
<td>repairing</td>
</tr>
<tr>
<td>T12</td>
<td>repairing</td>
<td>GuardCondition: RP::can_send() == true</td>
<td>repairing</td>
</tr>
<tr>
<td>T13</td>
<td>repairing</td>
<td>GuardCondition: RP::requested_changes() == &lt;empty&gt;</td>
<td>waiting</td>
</tr>
<tr>
<td>T14</td>
<td>ready</td>
<td>A new change was added to the RTPS Writer’s HistoryCache.</td>
<td>ready</td>
</tr>
<tr>
<td>T15</td>
<td>ready</td>
<td>A change was removed from the RTPS Writer’s HistoryCache.</td>
<td>ready</td>
</tr>
<tr>
<td>T16</td>
<td>any state</td>
<td>RTPS Writer is configured to no longer be matched with the RTPS Reader</td>
<td>final</td>
</tr>
</tbody>
</table>

### 8.4.9.2.1 Transition T1

This transition is triggered by the configuration of an RTPS Reliable StatefulWriter `the_rtps_writer` with a matching RTPS Reader. This configuration is done by the Discovery protocol (8.5) as a consequence of the discovery of a DDS DataReader that matches the DDS DataWriter that is related to `the_rtps_writer`.

The discovery protocol supplies the values for the ReaderProxy constructor parameters. The transition performs the following logical actions in the virtual machine:

```java
a_reader_proxy := new ReaderProxy( remoteReaderGuid, remoteGroupEntityId, expectsInlineQos, unicastLocatorList, multicastLocatorList);
the_rtps_writer.matched_reader_add(a_reader_proxy);
```

The ReaderProxy `a_reader_proxy` is initialized as discussed in 8.4.7.5. This includes initializing the set of unsent changes and applying a filter to each of the changes.

### 8.4.9.2.2 Transition T2

This transition is triggered by the guard condition [RP::unsent_changes() != <empty>] indicating that there are some changes in the RTPS Writer HistoryCache that have not been sent to the RTPS Reader represented by the ReaderProxy.
The transition performs no logical actions in the virtual machine.

8.4.9.2.3 Transition T3

This transition is triggered by the guard condition \([\text{RP::unsent\_changes()} == \text{false}]\) indicating that all changes in the RTPS Writer HistoryCache have been sent to the RTPS Reader represented by the ReaderProxy. Note that this does not indicate that the changes have been received, only that there has been an attempt made to send them.

The transition performs no logical actions in the virtual machine.

8.4.9.2.4 Transition T4

This transition is triggered by the guard condition \([\text{RP::can\_send()} == \text{true}]\) indicating that the RTPS Writer ‘the_rtps_writer’ has the resources needed to send a change to the RTPS Reader represented by the ReaderProxy ‘the_reader_proxy.’

The transition performs the following logical actions in the virtual machine:

\[
\text{a\_change\_seq\_num} := \text{the\_reader\_proxy.next\_unsent\_change();}
\]

\[
\text{if ( a\_change\_seq\_num > the\_reader\_proxy.highest\_sent\_seq\_num + 1 ) {}
\]

\[
\text{GAP} = \text{new GAP}(\text{the\_reader\_locator.highest\_sent\_seq\_num + 1,}
\]

\[
\text{a\_change\_seq\_num -1);}
\]

\[
\text{GAP.readerId} := \text{ENTITYID\_UNKNOWN;}
\]

\[
\text{GAP.filteredCount} := 0;
\]

\[
\text{send GAP;}
\]

\[
\}
\]

\[
\text{a\_change} := \text{the\_writer.writer\_cache.get\_change(a\_change\_seq\_num );}
\]

\[
\text{if ( DDS\_FILTER(}\text{the\_reader\_proxy, a\_change) ) {}
\]

\[
\text{DATA} = \text{new DATA}(a\_change);
\]

\[
\text{IF (the\_reader\_proxy.expectsInlineQos) {}
\]

\[
\text{DATA.inlineQos} := \text{the\_rtps\_writer.related\_dds\_writer.qos;}
\]

\[
\text{DATA.inlineQos} += a\_change.inlineQos;
\]

\[
\}
\]

\[
\text{DATA.readerId} := \text{ENTITYID\_UNKNOWN; // or ReaderProxy.entityId}
\]

\[
\text{send DATA;}
\]

\[
\}
\]

\[
\text{else {}
\]

\[
\text{GAP} = \text{new GAP}(a\_change.sequenceNumber);
\]

\[
\text{GAP.filteredCount} := 1;
\]

\[
\text{GAP.readerId} := \text{ENTITYID\_UNKNOWN; // or ReaderProxy.entityId}
\]

\[
\text{send GAP;}
\]

\[
\}
\]

\[
\text{the\_reader\_proxy.highest\_sent\_seq\_num} := \text{a\_change\_seq\_num;}
\]

The next unsent change ‘a_change’ present in the Writer Cache may not have a sequence number that matches the ReaderProxy (highest_sent_seq_num + 1). This may happen when a CacheChanges is removed from the Writer cache. For example, when using HISTORY QoS set to KEEP_LAST with depth == 1 and a new change for the same instance (key) replaces the previous one from the HistoryCache. In this case a GAP message is sent to indicate a range of sequence numbers not available to the Reader. This GAP represents CacheChanges that are not present in the Writer cache. Therefore, these changes do not appear counted in the GAP message filteredCount.

The next unsent change ‘a_change’ present in the Writer Cache may not pass a DDS_FILTER that indicates that the change is not relevant to that Reader. This may happen, for example, of the Reader specified a Content Filter. In this case a GAP is sent. However, this GAP represents CacheChanges that are present in the Writer cache but are not sent due to not being relevant. These changes do appear counted in the GAP message filteredCount.
The above logic is not meant to imply that each DATA or GAP Submessage is sent in a separate RTPS Message. Rather multiple Submessages can be combined into a single RTPS message.

The above illustrates the simplified case where a GAP Submessage includes a single sequence number. This would result in potentially many Submessages in cases where many sequence numbers in close proximity refer to changes that are not relevant to the Reader. Efficient implementations will try to combine multiple ‘not available’ sequence numbers into a single GAP message.

After the transition, the following post-conditions hold:

```
the_reader_proxy.highest_sent_seq_num == a_change_seq_num
```

### 8.4.9.2.5 Transition T5

This transition is triggered by the guard condition `RP::unacked_changes() == false` indicating that all changes in the RTPS Writer HistoryCache have been acknowledged by the RTPS Reader represented by the ReaderProxy.

The transition performs no logical actions in the virtual machine.

### 8.4.9.2.6 Transition T6

This transition is triggered by the guard condition `RP::unacked_changes() == true` indicating that there are changes in the RTPS Writer HistoryCache have not been acknowledged by the RTPS Reader represented by the ReaderProxy.

The transition performs no logical actions in the virtual machine.

### 8.4.9.2.7 Transition T7

This transition is triggered by the firing of a periodic timer configured to fire each W::heartbeatPeriod.

The transition performs the following logical actions for the StatefulWriter ‘the_rtps_writer’ in the virtual machine:

```
seq_num_min := the_rtps_writer.writer_cache.get_seq_num_min();
seq_num_max := the_rtps_writer.writer_cache.get_seq_num_max();
HEARTBEAT := new HEARTBEAT(the_rtps_writer.writerGuid, seq_num_min, seq_num_max);
HEARTBEAT.FinalFlag := NOT_SET;
HEARTBEAT.readerId := ENTITYID_UNKNOWN;
send HEARTBEAT;
```

### 8.4.9.2.8 Transition T8

This transition is triggered by the reception of an ACKNACK Message destined to the RTPS StatefulWriter ‘the_rtps_writer’ originating from the RTPS Reader represented by the ReaderProxy ‘the_reader_proxy.’ The transition performs the following logical actions in the virtual machine:

```
the_rtps_writer.acked_changes_set(ACKNACK.readerSNState.base - 1);
the_reader_proxy.requested_changes_set(ACKNACK.readerSNState.set);
```

After the transition the following post-conditions hold:

```
MIN { change.sequenceNumber IN the_reader_proxy.unacked_changes() } >=
ACKNACK.readerSNState.base - 1
```

### 8.4.9.2.9 Transition T9

This transition is triggered by the guard condition `RP::requested_changes() != <empty>` indicating that there are changes that have been requested by the RTPS Reader represented by the ReaderProxy.

The transition performs no logical actions in the virtual machine.
8.4.9.2.10 Transition T10

This transition is triggered by the reception of an ACKNACK message destined to the RTPS StatefulWriter ‘the_writer’ originating from the RTPS Reader represented by the ReaderProxy ‘the_reader_proxy.’

The transition performs the same logical actions as Transition T8 (8.4.9.2.8).

8.4.9.2.11 Transition T11

This transition is triggered by the firing of a timer indicating that the duration of $W::\text{nackResponseDelay}$ has elapsed since the state must_repair was entered.

The transition performs no logical actions in the virtual machine.

8.4.9.2.12 Transition T12

This transition is triggered by the guard condition $[\text{RP::can_send}() == \text{true}]$ indicating that the RTPS Writer ‘the_rtps_writer’ has the resources needed to send a change to the RTPS Reader represented by the ReaderProxy ‘the_reader_proxy.’

The transition performs the following logical actions in the virtual machine:

```java
a_change := the_reader_proxy.next_requested_change();
a_change.status := UNDERWAY;
cache_change := the_rtps_writer.writer_cache.find(a_change.sequence_number);
IF ( cache_change != NIL ) {
    IF ( DDS_FILTER(the_reader_proxy, a_change) ) {
        DATA = new DATA(cache_change, the_reader_proxy.remoteReaderGuid);
        IF (the_reader_proxy.expectsInlineQos) {
            DATA.inlineQos := the_rtps_writer.related_dds_writer.qos;
            DATA.inlineQos += cache_change.inlineQos;
        }
        send DATA;
    }
    ELSE {
        GAP = new GAP(a_change.sequenceNumber,
                       the_reader_proxy.remoteReaderGuid);
        GAP.filteredCount := 1;
        send GAP;
    }
}
ELSE {
    GAP = new GAP(a_change.sequenceNumber,
                   the_reader_proxy.remoteReaderGuid);
    GAP.filteredCount := 0;
    send GAP;
}
```

A requested change is identified by its sequence number. This change may still be in the Writer Cache or may have already been removed:

- If the change is still in the Writer cache the writer will check if it is relevant to the Reader (i.e. if passes any reader-specified filters and was not specifically directed to other readers). If the change is relevant the Writer will send a DATA message with the change information. If it is not relevant it will send a GAP message and account for the filtering in the GAP’s filteredCount.

- If the change is no longer in the Writer cache, the Writer will send a GAP and that change will not be counted in the GAP’s filteredCount.

The above logic is not meant to imply that each DATA or GAP Submessage is sent in a separate RTPS message. Rather multiple Submessages can be combined into a single RTPS message.

The above illustrates the simplified case where a GAP Submessage includes a single sequence number. This would result in potentially many Submessages in cases where many sequence numbers in close proximity refer
to changes that are not available to the Reader. Efficient implementations will combine multiple ‘not available’ sequence numbers as much as possible into a single GAP message.

After the transition the following post-condition holds:

\[
\text{a_change \ BELOONGS\-TO\ the\_reader\_proxy.requested\_changes() } \implies \text{FALSE}
\]

**8.4.9.2.13 Transition T13**

This transition is triggered by the guard condition \([\text{RP::requested\_changes() == <empty>}]\) indicating that there are no more changes requested by the RTPS Reader represented by the ReaderProxy.

The transition performs no logical actions in the virtual machine.

**8.4.9.2.14 Transition T14**

This transition is triggered by the addition of a new CacheChange ‘a_change’ to the HistoryCache of the RTPS Writer ‘the_rtps_writer’ by the corresponding DDS DataWriter.

The transition performs the following logical actions in the virtual machine:

\[
\text{ADD a_change TO the_rtps_writer.writer_cache}
\]

After the transition the following post-condition holds:

\[
\text{FOREACH proxy IN the_rtps_writer.matched_readers}
\text{proxy.unsent\_changes() == true}
\]

**8.4.9.2.15 Transition T15**

This transition is triggered by the removal of a CacheChange ‘a_change’ from the HistoryCache of the RTPS Writer ‘the_rtps_writer’ by the corresponding DDS DataWriter. For example, when using HISTORY QoS set to KEEP_LAST with depth == 1, a new change will cause the DDS DataWriter to remove the previous change for the same instance (key) from the HistoryCache.

**8.4.9.2.16 Transition T16**

This transition is triggered by the configuration of an RTPS Writer ‘the_rtps_writer’ to no longer be matched with the RTPS Reader represented by the ReaderProxy ‘the_reader_proxy.’ This configuration is done by the Discovery protocol (8.5) as a consequence of breaking a pre-existing match of a DDS DataReader with the DDS DataWriter related to ‘the_rtps_writer.’

The transition performs the following logical actions in the virtual machine:

\[
\text{the_rtps_writer.matched\_reader\_remove(the\_reader\_proxy);}
\text{delete the\_reader\_proxy;}
\]

**8.4.10 RTPS Reader Reference Implementations**

The RTPS Reader Reference Implementations are based on specializations of the RTPS Reader class, first introduced in 8.2. This sub clause describes the RTPS Reader and all additional classes used to model the RTPS Reader Reference Implementations. The actual behavior is described in 8.4.11 and 8.4.12.

**8.4.10.1 RTPS Reader**

RTPS Reader specializes RTPS Endpoint and represents the actor that receives CacheChange messages from one or more RTPS Writer endpoints. The Reference Implementations StatelessReader and StatefulReader specialize RTPS Reader and differ in the knowledge they maintain about the matched Writer endpoints.
The configuration attributes of the RTPS Reader are listed in Table 8.67 and allow for fine-tuning of the protocol behavior. The operations on an RTPS Reader are listed in Table 8.68.

### Table 8.67 - RTPS Reader configuration attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>meaning</th>
<th>relation to DDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>heartbeatResponseDelay</td>
<td>Duration_t</td>
<td>Protocol tuning parameter that allows the RTPS Reader to delay the sending of a positive or negative acknowledgment (see 8.4.12.2).</td>
<td>N/A</td>
</tr>
<tr>
<td>heartbeatSuppressionDuration</td>
<td>Duration_t</td>
<td>Protocol tuning parameter that allows the RTPS Reader to ignore HEARTBEATs that arrive 'too soon' after a previous HEARTBEAT was received.</td>
<td>N/A</td>
</tr>
<tr>
<td>reader_cache</td>
<td>History Cache</td>
<td>Contains the history of CacheChange changes for this RTPS Reader.</td>
<td>N/A</td>
</tr>
<tr>
<td>expectsInlineQos</td>
<td>bool</td>
<td>Specifies whether the RTPS Reader expects in-line QoS to be sent along with any data.</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.68 - RTPS Reader operations

<table>
<thead>
<tr>
<th>operation name</th>
<th>parameter list</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>new</td>
<td>&lt;return value&gt;</td>
<td>Reader</td>
</tr>
<tr>
<td></td>
<td>attribute_values</td>
<td>Set of attribute values required by the Reader and all the super classes.</td>
</tr>
</tbody>
</table>

The following sub clauses provide details on the operations.

8.4.10.1.1 Default Timing-Related Values

The following timing-related values are used as the defaults in order to facilitate ‘out-of-the-box’ interoperability between implementations.

```plaintext
heartbeatResponseDelay.sec = 0;
heartbeatResponseDelay.nanosec = 500 * 1000 * 1000; // 500 milliseconds
heartbeatSuppressionDuration.sec = 0;
heartbeatSuppressionDuration.nanosec = 0;
```

8.4.10.1.2 new

This operation creates a new RTPS Reader.

The newly-created reader ‘this’ is initialized as follows:

```plaintext
this.guid := <as specified in the constructor>;
this.unicastLocatorList := <as specified in the constructor>;
this.multicastLocatorList := <as specified in the constructor>;
this.reliabilityLevel := <as specified in the constructor>;
this.topicKind := <as specified in the constructor>;
this.expectsInlineQos := <as specified in the constructor>;
this.heartbeatResponseDelay := <as specified in the constructor>;
this.reader_cache := new HistoryCache;
```

8.4.10.2 RTPS StatelessReader

Specialization of RTPS Reader. The RTPS StatelessReader has no knowledge of the number of matched writers, nor does it maintain any state for each matched RTPS Writer.

In the current Reference Implementation, the StatelessReader does not add any configuration attributes or operations to those inherited from the Reader super class. Both classes are therefore identical. The virtual machine interacts with the StatelessReader using the operations in Table 8.69.
Table 8.69 - StatelessReader operations

<table>
<thead>
<tr>
<th>operation name</th>
<th>parameter list</th>
<th>parameter type</th>
</tr>
</thead>
<tbody>
<tr>
<td>new</td>
<td>&lt;return value&gt;</td>
<td>StatelessReader</td>
</tr>
<tr>
<td></td>
<td>attribute_values</td>
<td>Set of attribute values required by the StatelessReader and all the super classes.</td>
</tr>
</tbody>
</table>

8.4.10.2.1 new

This operation creates a new RTPS *StatelessReader*. The initialization is performed as on the RTPS *Reader* super class (8.4.10.1.2).

8.4.10.3 RTPS StatefulReader

Specialization of RTPS *Reader*. The RTPS *StatefulReader* keeps state on each matched RTPS *Writer*. The state kept on each writer is maintained in the RTPS *WriterProxy* class.

Table 8.70 - RTPS StatefulReader Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>meaning</th>
<th>relation to DDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>matched_writers</td>
<td>WriteProxy[*]</td>
<td>Used to maintain state on the remote Writers matched up with the Reader.</td>
<td>N/A</td>
</tr>
</tbody>
</table>

The virtual machine interacts with the *StatefulReader* using the operations in Table 8.71.

Table 8.71 - StatefulReader Operations

<table>
<thead>
<tr>
<th>operation name</th>
<th>parameter list</th>
<th>parameter type</th>
</tr>
</thead>
<tbody>
<tr>
<td>new</td>
<td>&lt;return value&gt;</td>
<td>StatefulReader</td>
</tr>
<tr>
<td></td>
<td>attribute_values</td>
<td>Set of attribute values required by the StatefulReader and all the super classes.</td>
</tr>
<tr>
<td>matched_writer_add</td>
<td>&lt;return value&gt;</td>
<td>void</td>
</tr>
<tr>
<td></td>
<td>a_writer_proxy</td>
<td>WriterProxy</td>
</tr>
<tr>
<td>matched_writer_remove</td>
<td>&lt;return value&gt;</td>
<td>void</td>
</tr>
<tr>
<td></td>
<td>a_writer_proxy</td>
<td>WriterProxy</td>
</tr>
<tr>
<td>matched_writer_lookup</td>
<td>&lt;return value&gt;</td>
<td>WriterProxy</td>
</tr>
<tr>
<td></td>
<td>a_writer_guid</td>
<td>GUID_t</td>
</tr>
</tbody>
</table>

8.4.10.3.1 new

This operation creates a new RTPS *StatefulReader*. The newly-created stateful reader ‘this’ is initialized as follows:

```
this.attributes := <as specified in the constructor>;
```
8.4.10.3.2 matched_writer_add

This operation adds the WriterProxy `a_writer_proxy` to the StatefulReader::matched_writers.

```c
ADD a_writer_proxy TO {this.matched_writers};
```

8.4.10.3.3 matched_writer_remove

This operation removes the WriterProxy `a_writer_proxy` from the set StatefulReader::matched_writers.

```c
DELETE a_writer_proxy;
```

8.4.10.3.4 matched_writer_lookup

This operation finds the WriterProxy with GUID_t `a_writer_guid` from the set StatefulReader::matched_writers.

```c
FIND proxy IN this.matched_writers SUCH-THAT (proxy.remoteWriterGuid == a_writer_guid);
return proxy;
```

8.4.10.4 RTPS WriterProxy

The RTPS WriterProxy represents the information an RTPS StatefulReader maintains on each matched RTPS Writer. The attributes of the RTPS WriterProxy are described in Table 8.72.

The association is a consequence of the matching of the corresponding DDS Entities as defined by the DDS specification, that is the DDS DataReader matching a DDS DataWriter by Topic, having compatible QoS, belonging to a common partition, and not being explicitly ignored by the application that uses DDS.

Table 8.72 - RTPS WriterProxy Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>meaning</th>
<th>relation to DDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>remoteWriterGuid</td>
<td>GUID_t</td>
<td>Identifies the matched Writer.</td>
<td>N/A Configured by discovery</td>
</tr>
<tr>
<td>unicastLocatorList</td>
<td>Locator_t[*]</td>
<td>List of unicast (address, port) combinations that can be used to send messages to the matched Writer or Writers. The list may be empty.</td>
<td>N/A Configured by discovery</td>
</tr>
<tr>
<td>multicastLocatorList</td>
<td>Locator_t[*]</td>
<td>List of multicast (address, port) combinations that can be used to send messages to the matched Writer or Writers. The list may be empty.</td>
<td>N/A Configured by discovery</td>
</tr>
<tr>
<td>dataMaxSizeSerialized</td>
<td>long</td>
<td>Optional attribute that indicates the maximum size of any <code>SerializedPayload</code> that may be sent by the matched Writer.</td>
<td>N/A Configured by discovery</td>
</tr>
<tr>
<td>changes_from_writer</td>
<td>CacheChange[*]</td>
<td>List of <code>CacheChange</code> changes received or expected from the matched RTPS Writer.</td>
<td>N/A Used to implement the behavior of the RTPS protocol</td>
</tr>
</tbody>
</table>
remoteGroupEntityId: EntityId_t
Identifies the group to which the matched Reader belongs

The EntityId of the Subscriber to which this DataReader belongs.

The virtual machine interacts with the **WriterProxy** using the operations in Table 8.73.

Table 8.73 - WriterProxy Operations

<table>
<thead>
<tr>
<th>operation name</th>
<th>parameter list</th>
<th>parameter type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>new</strong></td>
<td>&lt;return value&gt;</td>
<td>WriterProxy</td>
</tr>
<tr>
<td></td>
<td>attribute_values</td>
<td>Set of attribute values required by the WriterProxy.</td>
</tr>
<tr>
<td><strong>available_changes_max</strong></td>
<td>&lt;return value&gt;</td>
<td>SequenceNumber_t</td>
</tr>
<tr>
<td><strong>non_available_change_set</strong></td>
<td>&lt;return value&gt;</td>
<td>void</td>
</tr>
<tr>
<td></td>
<td>a_seq_num_seq</td>
<td>SequenceNumber_t[]</td>
</tr>
<tr>
<td></td>
<td>filteredCount</td>
<td>ChangeCount_t</td>
</tr>
<tr>
<td><strong>lost_changes_update</strong></td>
<td>&lt;return value&gt;</td>
<td>void</td>
</tr>
<tr>
<td></td>
<td>first_available_seq_num</td>
<td>SequenceNumber_t</td>
</tr>
<tr>
<td></td>
<td>changes_removed</td>
<td>boolean</td>
</tr>
<tr>
<td><strong>missing_changes</strong></td>
<td>&lt;return value&gt;</td>
<td>SequenceNumber_t[]</td>
</tr>
<tr>
<td><strong>missing_changes_update</strong></td>
<td>&lt;return value&gt;</td>
<td>void</td>
</tr>
<tr>
<td></td>
<td>last_available_seq_num</td>
<td>SequenceNumber_t</td>
</tr>
<tr>
<td><strong>received_change_set</strong></td>
<td>&lt;return value&gt;</td>
<td>void</td>
</tr>
<tr>
<td></td>
<td>a_seq_num</td>
<td>SequenceNumber_t</td>
</tr>
</tbody>
</table>

### 8.4.10.4.1 new

This operation creates a new RTPS **WriterProxy**.

The newly-created writer proxy ‘this’ is initialized as follows:

```plaintext
this.attributes := <as specified in the constructor>;
this.changes_from_writer := <all past and future samples from the writer>;
```

The **changes_from_writer** of the newly-created **WriterProxy** is initialized to contain all past and future samples from the **Writer** represented by the **WriterProxy**. This is a conceptual representation only, used to describe the Stateful Reference Implementation. The **ChangeFromWriter** status of each **CacheChange** in **changes_from_writer** is initialized to **UNKNOWN**, indicating the StatefulReader initially does not know whether any of these changes actually already exist. As discussed in 8.4.12.3, the status will change to **MISSING**, **RECEIVED**, **NOT_AVAILABLE** (NA_UNSPECIFIED, NA_FILTERED, or NA_REMOVED) as the StatefulReader is informed about their existence via a **HEARTBEAT** message, receives the actual changes via **DATA** or **DATA_FRAG** messages, or it is informed that the change is not available to the **Reader** and will not be delivered via a **GAP** or **HEARTBEAT** message.
8.4.10.4.2 available_changes_max

This operation returns the maximum SequenceNumber_t among the changes_from_writer changes in the RTPS WriterProxy that are available for access by the DDS DataReader.

The condition to make any CacheChange ‘a_change’ available for ‘access’ by the DDS DataReader is that there are no changes from the RTPS Writer with SequenceNumber_t smaller than or equal to a_change.sequenceNumber that have status MISSING or UNKNOWN. In other words, the available_changes_max and all previous changes are either RECEIVED or NOT_AVAILABLE.

Logical action in the virtual machine:

```
seq_num := MAX { change.sequenceNumber SUCH-THE
  ( change IN this.changes_from_writer
      AND ( change.status == RECEIVED
          OR change.status == NOT_AVAILABLE) ) }
```

8.4.10.4.3 not_available_change_set

This operation modifies the status of a ChangeFromWriter to indicate that the CacheChange with the SequenceNumber_t ‘a_seq_num’ is not available to the RTPS Reader.

The filteredCount parameter indicates the number of changes in the set that are still in the Writer cache but have been filtered for this Reader. The other changes are no longer present in the Writer cache so they should be considered as removed.

This operation may provide bulk notification on a set of changes, identified by their sequence numbers. In this case it may not be possible to determine whether a specific change was filtered or removed. That will happen if value of the filteredCount does not equal zero or the total number of changes in the set. If it is not possible to determine whether a change was filtered or removed, then the change status should be set to NA_UNSPECIFIED. Otherwise it should be set to NA_FILTERED or NA_REMOVED, as appropriate.

Logical action in the virtual machine:

```
FOREACH change FROM this.changes_from_writer SUCH-THE
  (change.sequenceNumber IN a_seq_num_set);
  IF ( filteredCount == COUNT(this.a_seq_num_set) ) THEN
    change.status := NA_FILTERED;
  ELSE IF ( filteredCount == 0)
    change.status := NA_REMOVED;
  ELSE
    change.status := NA_UNSPECIFIED;
```

8.4.10.4.4 lost_changes_update

This operation modifies the status stored in ChangeFromWriter for any changes in the WriterProxy whose status is MISSING or UNKNOWN and have sequence numbers lower than ‘first_available_seq_num.’ The status of those changes is modified to NA_REMOVED or NA_UNSPECIFIED, depending on the value of the parameter ‘changes_removed’. If ‘changes_removed’ is true, it indicates that the changes are no longer available in the WriterHistoryCache of the RTPS Writer represented by the RTPS WriterProxy.

Logical action in the virtual machine:

```
FOREACH change IN this.changes_from_writer
  SUCH-THE ( change.status == UNKNOWN OR change.status == MISSING
      AND seq_num < first_available_seq_num ) DO {
    change.status := NA_REMOVED;
}
```
8.4.10.4.5 missing_changes

This operation returns the subset of changes for the WriterProxy that have status ‘MISSING.’ The changes with status ‘MISSING’ represent the set of changes available in the HistoryCache of the RTPS Writer represented by the RTPS WriterProxy that have not been received by the RTPS Reader.

```
return { change IN this.changes_from_writer
         SUCH-TAT change.status == MISSING};
```

8.4.10.4.6 missing_changes_update

This operation modifies the status stored in ChangeFromWriter for any changes in the WriterProxy whose status is UNKNOWN and have sequence numbers smaller or equal to ‘last_available_seq_num.’ The status of those changes is modified from UNKNOWN to MISSING indicating that the changes are available at the WriterHistoryCache of the RTPS Writer represented by the RTPS WriterProxy but have not been received by the RTPS Reader.

Logical action in the virtual machine:
```
FOREACH change IN this.changes_from_writer
    SUCH-TAT ( change.status == UNKNOWN
                AND seq_num <= last_available_seq_num ) DO {
        change.status := MISSING;
    }
```

8.4.10.4.7 received_change_set

This operation modifies the status of the ChangeFromWriter that refers to the CacheChange with the SequenceNumber_t ‘a_seq_num.’ The status of the change is set to ‘RECEIVED,’ indicating it has been received. Logical action in the virtual machine:
```
FIND change FROM this.changes_from_writer
    SUCH-TAT change.sequenceNumber == a_seq_num;
    change.status := RECEIVED
```

8.4.10.5 RTPS ChangeFromWriter

The RTPS ChangeFromWriter is an association class that maintains information of a CacheChange in the RTPS Reader HistoryCache as it pertains to the RTPS Writer represented by the WriterProxy.

The attributes of the RTPS ChangeFromWriter are described in Table 8.74.

| Table 8.74 - RTPS ChangeFromWriter Attributes |

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>meaning</th>
<th>relation to DDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>ChangeFromWriter StatusKind</td>
<td>Indicates the status of a CacheChange relative to the RTPS Writer represented by the WriterProxy.</td>
<td>N/A. Used by the protocol.</td>
</tr>
<tr>
<td>is_relevant</td>
<td>bool</td>
<td>Indicates whether the change is relevant to the RTPS Reader.</td>
<td>The determination of relevant changes is affected by DDS DataReader TIME_BASED_FILTER QoS and also by the use of DDS ContentFilteredTopics.</td>
</tr>
</tbody>
</table>

The type ChangeFromWriter StatusKind is an enumeration that can take the following values:
• UNKNOWN. This means that the Reader has not received the change and it is not known if the change is in the WriterCache. It may be not have been written yet (i.e. is a change potentially in the future), it may be in the WriterCache, or it may have been written and removed from the WriterCache.

• MISSING. This means that the Reader has been informed that the Writer has potentially this change in its WriterCache so the Reader can request it.

• RECEIVED. This means that the Reader has received the change via DATA or DATA_FRAG messages.

• NOT_AVAILABLE. This means that the Reader has been informed that the change will not be sent by the Writer. There are three possible sub-statues:
  o NA_FILTERED. This the change is in the WriterCache but it will not be sent to the Reader because the Writer has filtered it for the Reader.
  o NA_REMOVED. This means the change was at some point in the WriterCache but it is no longer there and therefore it will not be delivered to the Reader.
  o NA_UNSPECIFIED. This means that the Writer did not provide enough information for the Reader to determine the reason why the change is not available to the Reader.

8.4.11 RTPS StatelessReader Behavior

8.4.11.1 Best-Effort StatelessReader Behavior

The behavior of the Best-Effort RTPS StatelessReader is independent of any writers and is described in Figure 8.22.

![Diagram of RTPS StatelessReader behavior]

Figure 8.22 - Behavior of the Best-Effort StatelessReader

The state-machine transitions are listed in Table 8.75.

Table 8.75 - Transitions for Best-effort StatelessReader behavior

<table>
<thead>
<tr>
<th>Transition</th>
<th>state</th>
<th>event</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>initial</td>
<td>RTPS Reader is created</td>
<td>waiting</td>
</tr>
<tr>
<td>T2</td>
<td>waiting</td>
<td>DATA message is received</td>
<td>waiting</td>
</tr>
<tr>
<td>T3</td>
<td>waiting</td>
<td>RTPS Reader is deleted</td>
<td>final</td>
</tr>
</tbody>
</table>

8.4.11.1.1 Transition T1

This transition is triggered by the creation of an RTPS StatelessReader ‘the_rtps_reader.’ This is the result of the creation of a DDS DataReader as described in 8.2.10.
The transition performs no logical actions in the virtual machine.

8.4.11.1.2 Transition T2

This transition is triggered by the reception of a DATA message by the RTPS Reader ‘the_rtps_reader.’ The DATA message contains the change ‘a_change.’ The representation is described in 8.3.8.2.

The stateless nature of the StatelessReader prevents it from maintaining the information required to determine the highest sequence number received so far from the originating RTPS Writer. The consequence is that in those cases the corresponding DDS DataReader may be presented duplicate or out-of-order changes. Note that if the DDS DataReader is configured to order data by ‘source timestamp,’ any available data will still be presented in-order when accessing the data through the DDS DataReader.

As mentioned in 8.4.3, actual stateless implementations may try to avoid this limitation and maintain this information in non-permanent fashion (using for example a cache that expires information after a certain time) to approximate, to the extent possible, the behavior that would result if the state were maintained.

The transition performs the following logical actions in the virtual machine:

```java
    a_change := new CacheChange(DATA);
    the_rtps_reader.reader_cache.add_change(a_change);
```

8.4.11.1.3 Transition T3

This transition is triggered by the destruction of an RTPS Reader ‘the_rtps_reader.’ This is the result of the destruction of a DDS DataReader as described in 8.2.10.

The transition performs no logical actions in the virtual machine.

8.4.11.2 Reliable StatelessReader Behavior

This combination is not supported by the RTPS protocol. In order to implement the reliable protocol, the RTPS Reader must keep some state on each matched RTPS Writer.

8.4.12 RTPS StatefulReader Behavior

8.4.12.1 Best-Effort StatefulReader Behavior

The behavior of the Best-Effort RTPS StatefulReader with respect to each matched Writer is described in Figure 8.23.
Figure 8.23 - Behavior of the Best-Effort StatefulReader with respect to each matched Writer

The state-machine transitions are listed in Table 8.76.

Table 8.76 - Transitions for Best-Effort StatefulReader behavior with respect to each matched writer

<table>
<thead>
<tr>
<th>Transition</th>
<th>state</th>
<th>event</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>initial</td>
<td>RTPS Reader is configured with a matched RTPS Writer</td>
<td>waiting</td>
</tr>
<tr>
<td>T2</td>
<td>waiting</td>
<td>DATA message is received from the matched Writer</td>
<td>waiting</td>
</tr>
<tr>
<td>T3</td>
<td>waiting</td>
<td>RTPS Reader is configured to no longer be matched with the RTPS Writer</td>
<td>final</td>
</tr>
<tr>
<td>T4</td>
<td>waiting</td>
<td>GAP message is received</td>
<td>waiting</td>
</tr>
</tbody>
</table>

8.4.12.1.1 Transition T1

This transition is triggered by the configuration of an RTPS Reader ‘the_rtps_reader’ with a matching RTPS Writer. This configuration is done by the Discovery protocol (8.5) as a consequence of the discovery of a DDS DataWriter that matches the DDS DataReader that is related to ‘the_rtps_reader.’

The discovery protocol supplies the values for the WriterProxy constructor parameters. The transition performs the following logical actions in the virtual machine:

```
a_writer_proxy := new WriterProxy(remoteWriterGuid, remoteGroupEntityId, unicastLocatorList, multicastLocatorList);
the_rtps_reader.matched_writer_add(a_writer_proxy);
```

The WriterProxy is initialized with all past and future samples from the Writer as discussed in 8.4.10.4.
8.4.12.1.2 Transition T2

This transition is triggered by the reception of a DATA message by the RTPS Reader ‘the_rtps_reader.’ The DATA message contains the change ‘a_change.’ The representation is described in 8.3.8.2.

The Best-Effort reader checks that the sequence number associated with the change is strictly greater than the highest sequence number of all changes received in the past from this RTPS Writer (WP::available_changes_max()). If this check fails, the RTPS Reader discards the change. This ensures that there are no duplicate changes and no out-of-order changes.

The transition performs the following logical actions in the virtual machine:

```plaintext
a_change := new CacheChange(DATA);
writer_guid := {Receiver.SourceGuidPrefix, DATA.writerId};
writer_proxy := the_rtps_reader.matched_writer_lookup(writer_guid);
expected_seq_num := writer_proxy.available_changes_max() + 1;
if ( a_change.sequenceNumber >= expected_seq_num ) {
    the_rtps_reader.reader_cache.add_change(a_change);
    writer_proxy.received_change_set(a_change.sequenceNumber);
    if ( a_change.sequenceNumber > expected_seq_num ) {
        writer_proxy.lost_changes_update(a_change.sequenceNumber, false);
    }
}
```

After the transition the following post-conditions hold:

```
writer_proxy.available_changes_max() >= a_change.sequenceNumber
```

8.4.12.1.3 Transition T3

This transition is triggered by the configuration of an RTPS Reader ‘the_rtps_reader’ to no longer be matched with the RTPS Writer represented by the WriterProxy ‘the_writer_proxy.’ This configuration is done by the Discovery protocol (8.5) as a consequence of breaking a pre-existing match of a DDS DataWriter with the DDS DataReader related to ‘the_rtps_reader.’

The transition performs the following logical actions in the virtual machine:

```
the_rtps_reader.matched_writer_remove(the_writer_proxy);
delete the_writer_proxy;
```

8.4.12.1.4 Transition T4

This transition is triggered by reception of a GAP message destined to the RTPS StatefulReader ‘the_reader’ originating from the RTPS Writer represented by the WriterProxy ‘the_writer_proxy’.

The transition performs the following logical actions in the virtual machine:

```
the_writer_proxy.not_available_change_set(GAP. sequence_number_set, GAP.filteredCount);
```

8.4.12.2 Reliable StatefulReader Behavior

The behavior of the Reliable RTPS StatefulReader with respect to each matched RTPS Writer is described in Figure 8.24.
Figure 8.24 - Behavior of the Reliable StatefulReader with respect to each matched Writer

The state-machine transitions are listed in Table 8.77.

Table 8.77 - Transitions for Reliable reader behavior with respect to a matched writer

<table>
<thead>
<tr>
<th>Transition</th>
<th>state</th>
<th>event</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>initial1</td>
<td>RTPS Reader is configured with a matched RTPS Writer.</td>
<td>waiting</td>
</tr>
<tr>
<td>T2</td>
<td>waiting</td>
<td>HEARTBEAT message is received.</td>
<td>if (HB.FinalFlag == NOT_SET) then must_send_ack else if (HB.LivelinessFlag == NOT_SET) then may_send_ack else waiting</td>
</tr>
</tbody>
</table>
| T3         | may_send_ack | GuardCondition:  
  WP::missing_changes() == <empty>                                  | waiting    |
| T4         | may_send_ack | GuardCondition:  
  WP::missing_changes() != <empty>                                  | must_send_ack |
| T5         | must_send_ack | after(R::heartbeatResponseDelay)                                     | waiting    |
T6  initial2  RTPS Reader is configured with a matched RTPS Writer. ready
T7  ready  HEARTBEAT message is received. ready
T8  ready  DATA message is received. ready
T9  ready  GAP message is received. ready
T10  any state  RTPS Reader is configured to no longer be matched with the RTPS Writer. final

8.4.12.2.1 Transition T1

This transition is triggered by the configuration of an RTPS Reliable StatefulReader ‘the_rtps_reader’ with a matching RTPS Writer. This configuration is done by the Discovery protocol (8.5) as a consequence of the discovery of a DDS DataWriter that matches the DDS DataReader that is related to ‘the_rtps_reader.’

The discovery protocol supplies the values for the WriterProxy constructor parameters. The transition performs the following logical actions in the virtual machine:

```
a_writer_proxy := new WriterProxy(remoteWriterGuid, remoteGroupEntityId, unicastLocatorList, multicastLocatorList);
the_rtps_reader.matched_writer_add(a_writer_proxy);
```

The WriterProxy is initialized with all past and future samples from the Writer as discussed in 8.4.10.4.

8.4.12.2 Transition T2

This transition is triggered by the reception of a HEARTBEAT message destined to the RTPS StatefulReader ‘the_reader’ originating from the RTPS Writer represented by the WriterProxy ‘the_writer_proxy.’

The transition performs no logical actions in the virtual machine. Note however that the reception of a HEARTBEAT message causes the concurrent transition T7 (8.4.12.2.7), which performs logical actions.

8.4.12.3 Transition T3

This transition is triggered by the guard condition [W::missing_changes() == <empty>] indicating that all changes known to be in the HistoryCache of the RTPS Writer represented by the WriterProxy have been received by the RTPS Reader.

The transition performs no logical actions in the virtual machine.

8.4.12.4 Transition T4

This transition is triggered by the guard condition [W::missing_changes() != <empty>] indicating that there are some changes known to be in the HistoryCache of the RTPS Writer represented by the WriterProxy, which have not been received by the RTPS Reader.

The transition performs no logical actions in the virtual machine.

8.4.12.5 Transition T5

This transition is triggered by the firing of a timer indicating that the duration of R::heartbeatResponseDelay has elapsed since the state must_send_ack was entered.

The transition performs the following logical actions for the WriterProxy ‘the_writer_proxy’ in the virtual machine:

```
missing_seq_num_set.base := the_writer_proxy.available_changes_max() + 1;
missing_seq_num_set.set := <empty>;
```
FOREACH change IN the_writer_proxy.missing_changes() DO
ADD change.sequenceNumber TO missing_seq_num_set.set;
send ACKNACK(missing_seq_num_set);

The above logical action does not express the fact that the PSM mapping of the ACKNACK message will be limited in its capacity to contain sequence numbers. In the case where the ACKNACK message cannot accommodate the complete list of missing sequence numbers it should be constructed such that it contains the subset with smallest value of the sequence number.

8.4.12.2.6 Transition T6

Similar to T1 (8.4.12.2.1), this transition is triggered by the configuration of an RTPS Reliable StatefulReader ‘the_rtps_reader’ with a matching RTPS Writer.

The transition performs no logical actions in the virtual machine.

8.4.12.2.7 Transition T7

This transition is triggered by the reception of a HEARTBEAT message destined to the RTPS StatefulReader ‘the_reader’ originating from the RTPS Writer represented by the WriterProxy ‘the_writer_proxy.’

The firstSN in the HEARTBEAT message indicates the lowest sequence number in the Writer cache. For this reason, the call to lost_changes_updates sets the parameter removed_samples to ‘true’.

The transition performs the following logical actions in the virtual machine:
the_writer_proxy.missing_changes_update(HEARTBEAT.lastSN);
the_writer_proxy.lost_changes_update(HEARTBEAT.firstSN, true);

8.4.12.2.8 Transition T8

This transition is triggered by the reception of a DATA message destined to the RTPS StatefulReader ‘the_reader’ originating from the RTPS Writer represented by the WriterProxy ‘the_writer_proxy.’

The transition performs the following logical actions in the virtual machine:
a_change := new CacheChange(DATA);
the_reader.reader_cache.add_change(a_change);
the_writer_proxy.received_change_set(a_change.sequenceNumber);

Any filtering is done when accessing the data using the DDS DataReader read or take operations, as described in 8.2.10.

8.4.12.2.9 Transition T9

This transition is triggered by the reception of a GAP message destined to the RTPS StatefulReader ‘the_reader’ originating from the RTPS Writer represented by the WriterProxy ‘the_writer_proxy.’

The transition performs the following logical actions in the virtual machine:
the_writer_proxy.not_available_change_set(GAP.sequence_number_set, GAP.filteredCount);

8.4.12.2.10 Transition T10

This transition is triggered by the configuration of an RTPS Reader ‘the_rtps_reader’ to no longer be matched with the RTPS Writer represented by the WriterProxy ‘the_writer_proxy.’ This configuration is done by the Discovery protocol (8.5) as a consequence of breaking a pre-existing match of a DDS DataWriter with the DDS DataReader related to ‘the_rtps_reader.’

The transition performs the following logical actions in the virtual machine:
the_rtps_reader.matched_writer_remove(the_writer_proxy);
delete the_writer_proxy;
8.4.12.3 ChangeFromWriter illustrated

The ChangeFromWriter keeps track of the communication status (attribute status) and relevance (attribute is_relevant) of each CacheChange with respect to a specific remote RTPS Writer.

The behavior of the RTPS StatefulReader described in Figure 8.24 modifies each ChangeFromWriter as a side-effect of the operation of the protocol. To further define the protocol, it is illustrative to examine the State Machine representing the value of the status attribute for any given ChangeFromWriter. This is shown in Figure 8.25 for a Reliable StatefulReader. A Best-Effort StatefulReader uses only a subset of the state-diagram.

Figure 8.25 - Changes in the value of the status attribute of each ChangeFromWriter

The states have the following meanings:

- **<Unknown>**: A CacheChange with SequenceNumber_t seq_num may or may not be available yet at the RTPS Writer.
- **<Missing>**: The CacheChange with SequenceNumber_t seq_num is available in the RTPS Writer and has not been received yet by the RTPS Reader.
- **<NotRequested>**: The CacheChange with SequenceNumber_t seq_num has not been requested from the RTPS Writer, no response is expected.
- **<Requested>**: The CacheChange with SequenceNumber_t seq_num was requested from the RTPS Writer, a response might be pending or underway.
- **<Received>**: The CacheChange with SequenceNumber_t seq_num was received: as a DATA.
- **<NotAvailable>**: The CacheChange with SequenceNumber_t seq_num is no longer available at the RTPS Writer. It will not be received. The Reader has received this information via a HEARTBEAT or a GAP without an indication that the change was filtered. There are 3 substates:
  - **<Filtered>**: The CacheChange with SequenceNumber_t seq_num was received as a GAP with an indication that the change was filtered by the Writer for this reader.
  - **<Removed>**: The CacheChange with SequenceNumber_t seq_num was received as a GAP with an indication that the change was not filtered by the Writer. Or it was received as a HEARTBEAT and the Reader is reliable.
  - **<Unspecified>**: The CacheChange with SequenceNumber_t seq_num was received as a GAP with not enough information to determine whether it was filtered or not.
The following describes the main events that trigger transitions in the State Machine. Note that this state-machine just keeps track of the ‘status’ attribute of a particular ChangeForReader and does not perform any specific actions nor send any messages.

- new ChangeFromWriter(seq_num): The WriterProxy has created a ChangeFromWriter association class to track the state of a CacheChange with SequenceNumber_t seq_num.
- received HB(firstSN <= seq_num <= lastSN): The Reader has received a HEARTBEAT with HEARTBEAT.firstSN <= seq_num <= HEARTBEAT.lastSN, indicating a CacheChange with that sequence number is available from the RTPS Writer.
- sent NACK(seq_num): The Reader has sent an ACKNACK message containing the seq_num inside the ACKNACK.readerSNState, indicating the RTPS Reader has not received the CacheChange and is requesting it is sent again.
- received GAP(seq_num): The Reader has received a GAP message where seq_num is inside GAP.gapList, which means that the seq_num will not be sent to the RTPS Reader.
- The GAP may contain an indication that the cache change is still available in the Writer but was filtered for the Reader.
- received DATA(seq_num): The Reader has received a DATA message with DATA.sequenceNumber == seq_num.
- received HB(firstSN > seq_num): The Reader has received a HEARTBEAT with HEARTBEAT.firstSN > seq_num, indicating the CacheChange with that sequence number is no longer present in the RTPS Writer cache.

8.4.13 Writer Liveliness Protocol

The DDS specification requires the presence of a liveliness mechanism. RTPS realizes this requirement with the Writer Liveliness Protocol. The Writer Liveliness Protocol defines the required information exchange between two Participants in order to assert the liveliness of Writers contained by the Participants.

All implementations must support the Writer Liveliness Protocol in order to be interoperable.

8.4.13.1 General Approach

The Writer Liveliness Protocol uses pre-defined built-in Endpoints. The use of built-in Endpoints means that once a Participant knows of the presence of another Participant, it can assume the presence of the built-in Endpoints made available by the remote Participant and establish the association with the locally matching built-in Endpoints.

The protocol used to communicate between built-in Endpoints is the same as used for application-defined Endpoints.

8.4.13.2 Built-in Endpoints Required by the Writer Liveliness Protocol

The built-in Endpoints required by the Writer Liveliness Protocol are the BuiltinParticipantMessageWriter and BuiltinParticipantMessageReader. The names of these Endpoints reflect the fact that they are general-purpose. These Endpoints are used for liveliness but can be used for other data in the future.

The RTPS Protocol reserves the following values of the EntityId_t for these built-in Endpoints:

ENTITYID_P2P_BUILTIN_PARTICIPANT_MESSAGE_WRITER
ENTITYID_P2P_BUILTIN_PARTICIPANT_MESSAGE_READER

The actual value for each of these EntityId_t instances is defined by each PSM.

8.4.13.3 BuiltinParticipantMessageWriter and BuiltinParticipantMessageReader QoS

For interoperability, both the BuiltinParticipantMessageWriter and BuiltinParticipantMessageReader shall use the following QoS values:

- durability.kind = TRANSIENT_LOCAL_DURABILITY
- history.kind = KEEP_LAST_HISTORY_QOS
• history.depth = 1

The BuiltinParticipantMessageWriter shall use reliability.kind = RELIABLE_RELIABILITY_QOS.

The BuiltinParticipantMessageReader may be configured to use either RELIABLE_RELIABILITY_QOS or BEST_EFFORT_RELIABILITY_QOS. If the BuiltinParticipantMessageReader is configured to use BEST_EFFORT_RELIABILITY_QOS then the BEST_EFFORT_PARTICIPANT_MESSAGE_DATA_READER flag in ParticipantProxy::builtinEndpointQos shall be set.

If the ParticipantProxy::builtinEndpointQos is included in the SPDPdiscoveredParticipantData, then the BuiltinParticipantMessageWriter shall treat the BuiltinParticipantMessageReader as indicated by the flags. If the ParticipantProxy::builtinEndpointQos is not included then the BuiltinParticipantMessageWriter shall treat the BuiltinParticipantMessageReader as if it is configured with RELIABLE_RELIABILITY_QOS.

8.4.13.4 Data Types Associated with Built-in Endpoints used by Writer Liveliness Protocol

Each RTPS Endpoint has a HistoryCache that stores changes to the data-objects associated with the Endpoint. This is also true for the RTPS built-in Endpoints. Therefore, each RTPS built-in Endpoint depends on some DataType that represents the logical contents of the data written into its HistoryCache.

Figure 8.26 defines the ParticipantMessageData datatype associated with the RTPS built-in Endpoint for the DCPSParticipantMessage Topic.

![ParticipantMessageData](image)

Figure 8.26 - Participant Message Data

8.4.13.5 Implementing Writer Liveliness Protocol Using the BuiltinParticipantMessageWriter and Builtin-ParticipantMessageReader

The liveliness of a subset of Writers belonging to a Participant is asserted by writing a sample to the BuiltinParticipantMessageWriter. If the Participant contains one or more Writers with a liveliness of AUTOMATIC_LIVELINESS_QOS, then one sample is written at a rate faster than the smallest lease duration among the Writers sharing this QoS. Similarly, a separate sample is written if the Participant contains one or more Writers with a liveliness of MANUAL_BY_PARTICIPANT_LIVELINESS_QOS at a rate faster than the smallest lease duration among these Writers. The two instances are orthogonal in purpose so that if a Participant contains Writers of each of the two liveliness kinds described, two separate instances must be periodically written. The instances are distinguished using their DDS key, which is comprised of the participantGuidPrefix and kind fields. Each of the two types of liveliness QoS handled through this protocol will result in a unique kind field and therefore form two distinct instances in the HistoryCache.

In both liveliness cases the participantGuidPrefix field contains the GuidPrefix_t of the Participant that is writing the data (and therefore asserting the liveliness of its Writers).

The DDS liveliness kind MANUAL_BY_TOPIC_LIVELINESS_QOS is not implemented using the BuiltinParticipantMessageWriter and BuiltinParticipantMessageReader. It is discussed in 8.7.2.2.3.

8.4.14 Optional Behavior

This sub clause describes optional features of the RTPS protocol. Optional features may not be supported by all RTPS implementations. An optional feature does not affect basic interoperability, but is only available if all implementations involved support it.
8.4.14.1 Large Data

As described in 7.6, RTPS poses very few requirements on the underlying transport. It is sufficient that the transport offers a connectionless service capable of sending packets best-effort.

That said, a transport may impose its own limitations. For example, it may limit the maximum packet size (e.g., 64K for UDP) and hence the maximum RTPS Submessage size. This mainly affects the Data Submessage, as it limits the maximum size of the serializedData or also, the maximum serialized size of the data type used.

In order to address this limitation, 8.3.8 introduces the following Submessages to enable fragmenting large data:

- DataFrag
- HeartbeatFrag
- NackFrag

The following sub clauses list the corresponding behavior required for interoperability.

8.4.14.1.1 How to select the fragment size

The fragment size is determined by the Writer and must meet the following requirements:

- All transports available to the Writer must be able to accommodate DataFrag Submessages containing at least one fragment. This means the transport with the smallest maximum message size determines the fragment size.
- The fragment size must be fixed for a given Writer and is identical for all remote Readers. By fixing the fragment size, the data a fragment number refers to does not depend on a particular remote Reader. This simplifies processing negative acknowledgements (NackFrag) from a Reader.
- The fragment size must satisfy: fragment size <= 65536 bytes.

Note the fragment size is determined by all transports available to the Writer, not simply the subset of transports required to reach all currently known Readers. This ensures newly discovered Readers, regardless of the transport they can be reached on, can be accommodated without having to change the fragment size, which would violate the above requirements.

8.4.14.1.2 How to send fragments

If fragmentation is required, a Data Submessage is replaced by a sequence of DataFrag Submessages. The protocol behavior for sending DataFrag Submessages matches that for sending regular Data Submessages with the following additional requirements:

- DataFrag Submessages are sent in order, where ordering is defined by increasing fragment numbers. Note this does not guarantee in order arrival.
- Data must only be fragmented if required. If multiple transports are available to the Writer and some transports do not require fragmentation, a regular Data Submessage must be sent on those transports instead. Likewise, for variable size data types, a regular Data Submessage must be used if fragmentation is not required for a particular sequence number.
- For a given sequence number, if in-line QoS parameters are used, they must be included with the first DataFrag Submessage (containing the fragment with fragment number equal to 1). They may also be included with subsequent DataFrag submessages for this sequence number, but this is not required.

If a transport can accommodate multiple fragments of the given fragment size, it is recommended that implementations concatenate as many fragments as possible into a single DataFrag message.
When sending multiple DataFrag messages, flow control may be required to avoid flooding the network. Possible approaches include a leaky bucket or token bucket flow control scheme. This is not part of the RTPS specification.

8.4.14.1.3 How to re-assemble fragments

DataFrag Submessages contain all required information to re-assemble the serialized data. Once all fragments have been received, the same protocol behavior applies as for a regular Data Submessage.

Note that implementations must be able to handle out-of-order arrival of DataFrag submessages.

8.4.14.1.4 Reliable Communication

The protocol behavior for reliably sending DataFrag Submessages matches that for sending regular Data Submessages with the following additional requirements:

- The semantics for a Heartbeat Submessage remains unchanged: A Heartbeat message must only include those sequence numbers for which all fragments are available.

- The semantics for an AckNack Submessage remain unchanged: an AckNack message must only positively acknowledge a sequence number when all fragments were received for that sequence number. Likewise, a sequence number must be negatively acknowledged only when all fragments are missing.

- In order to negatively acknowledge a subset of fragments for a given sequence number, a NackFrag Submessage must be used. When data is fragmented, a Heartbeat may trigger both AckNack and NackFrag Submessages.

Additional considerations:

- As mentioned above, a Heartbeat Submessage can only include a sequence number once all fragments for that sequence number are available. If a Writer wants to inform a Reader on the partial availability of fragments for a given sequence number, a HeartbeatFrag Submessage can be used instead. Fragment level reliability may be helpful for very large data and when using flow control.

- A NackFrag Submessage can only be sent in response to a Heartbeat or HeartbeatFrag submessage.

8.4.15 Implementation Guidelines

The contents of this sub clause are not part of the formal specification of the protocol. The purpose of this sub clause is to provide guidelines for high-performance implementations of the protocol.

8.4.15.1 Implementation of ReaderProxy and WriterProxy

The PIM models the ReaderProxy as maintaining an association with each CacheChange in the Writer’s HistoryCache. This association is modeled as being mediated by the association class ChangeForReader. The direct implementation of this model would result in a lot of information being maintained for each ReaderProxy. In practice, what is required is that the ReaderProxy is able to implement the operations used by the protocol and this does not require the use of explicit associations.

For example, the operations unsent_changes() and next_unsent_change() can be implemented by having the ReaderProxy maintain a single sequence number ‘highestSeqNumSent.’ The highestSeqNumSent would record the highest value of the sequence number of any CacheChange sent to the ReaderProxy. Using this the operation unsent_changes() could be implemented by looking up all changes in the HistoryCache and selecting the ones with sequenceNumber greater than highestSeqNumSent. The implementation of next_unsent_change() would also look at the HistoryCache and return the CacheChange that has the next-highest sequence number.
greater than highestSeqNumSent. These operations could be done efficiently if the HistoryCache maintains an index by sequenceNumber.

The same techniques can be used to implement, requested_changes(), requested_changes_set(), and next_requested_change(). In this case, the implementation can maintain a sliding window of sequence numbers (which can be efficiently represented by a SequenceNumber_t lowestRequestedChange and a fixed-length bitmap) to store whether a particular sequence number is currently requested. Requests that do not fit in the window can be ignored as they correspond to sequence numbers higher than the ones in the window and the reader can be relied on resending the request later if it is still missing the change.

Similar techniques can be used to implement acked_changes_set() and unacked_changes().

### 8.4.15.2 Efficient use of Gap and AckNack Submessages

Both Gap and AckNack Submessages are designed such that they can contain information about a set of sequence numbers. For simplicity, the virtual machine used in the protocol description did not always attempt to fully use these Submessages to store all the sequence numbers for which they would apply. The result would be that sometimes multiple Gap or AckNack messages would be sent when, a more efficient implementation, would have combined these Submessages into a single one. All these implementations are compliant with the protocol and interoperable. However, implementations that combine multiple Gap and AckNack Submessages and take advantage of the ability of these Submessages to contain a set of sequence number will be more efficient in both bandwidth and CPU usage.

### 8.4.15.3 Coalescing multiple Data Submessages

The RTPS protocol allows multiple Submessages to be coalesced into a single RTPS message. This means that they will all share a single RTPS Header and be sent in a single ‘network-transport transaction.’ Most network-transports have a relatively-large fixed overhead compared with the extra cost of additional bytes in the message. Therefore, implementations that combine Submessages into a single RTPS message will in general make better utilization of CPU and bandwidth.

A particularly common case is the coalescing of multiple Data Submessages into a single RTPS message. The need for this can occur in a response to an AckNack requesting multiple changes or as a result of multiple changes made on the writer side that have not yet been propagated to the reader. In all these cases, it is generally beneficial to coalesce the Submessages into fewer RTPS messages.

Note that the coalescing of Data Submessages is not restricted to Submessages originating from the same RTPS Writer. It is also possible to coalesce Submessages originating from multiple RTPS Writer entities. RTPS Writer entities that correspond to DDS DataWriter entities belonging to the same DDS Publisher are prime candidates for this.

### 8.4.15.4 Piggybacking HeartBeat Submessages

The RTPS protocol allows Submessages of different kinds to be coalesced into a single RTPS message. A particularly useful case is the piggybacking of HeartBeat Submessages following Data Submessages. This allows the RTPS Writer to explicitly request an acknowledgment of the changes it sent without the additional traffic needed to send a separate HeartBeat.

### 8.4.15.5 Sending to unknown readerId

As described in the Messages Module, it is possible to send RTPS Messages where the readerId is left unspecified (ENTITYID_UNKNOWN). This is required when sending these Messages over Multicast, but also allows to send a single Message over unicast to reach multiple Readers within the same Participant. Implementations are encouraged to use this feature to minimize bandwidth usage.
8.4.15.6 Reclaiming Finite Resources from Unresponsive Readers

An implementation likely has finite resources to work with. For a Writer, reclaiming queue resources should happen when all Readers have acknowledged a sample in the queue and resources limits dictate that the old sample entry is to be used for a new sample.

There may be scenarios where an alive Reader becomes unresponsive and will never acknowledge the Writer. Instead of blocking on the unresponsive Reader, the Writer should be allowed to deem the Reader as ‘Inactive’ and proceed in updating its queue. The state of a Reader is either Active or Inactive. Active Readers have sent ACKNACKs that have been recently received. The Writer should determine the inactivity of a Reader by using a mechanism based on the rate and number of ACKNACKs received. Then samples that have been acknowledged by all Active Readers can be freed, and the Writer can reclaim those resources if necessary. Note that strict reliability is not guaranteed when a Reader becomes Inactive.

8.4.15.7 Setting Count in Heartbeat, HeartbeatFrag, AckNack, and NackFrag submessages

The Count element of a HEARTBEAT differentiates between logical HEARTBEATs. A received HEARTBEAT with the same Count as a previously received HEARTBEAT can be ignored to prevent triggering a duplicate repair session. So, an implementation should ensure that same logical HEARTBEATs are tagged with the same Count.

The HEARTBEATS received by a Reader should have Counts greater than all older HEARTBEATs from the same Writer. Otherwise they can be discarded. As long as this requirement is met, it is up to the implementation to decide whether a Writer keeps a Count specific to each Reader or the Count is shared among all of its matching Readers. The same logic applies for Counts of ACKNACKs. It is up to the implementation to decide whether a Reader keeps a Count specific to each Writer or if it is shared among all of its matching Writers.

The Count element should be incremented and compared according to modular arithmetic rules in order to accommodate the integer overflow.

8.5 Discovery Module

The RTPS Behavior Module assumes RTPS Endpoints are properly configured and paired up with matching remote Endpoints. It does not make any assumptions on how this configuration took place and only defines how to exchange data between these Endpoints.

In order to be able to configure Endpoints, implementations must obtain information on the presence of remote Endpoints and their properties. How to obtain this information is the subject of the Discovery Module.

The Discovery Module defines the RTPS discovery protocol. The purpose of the discovery protocol is to allow each RTPS Participant to discover other relevant Participants and their Endpoints. Once remote Endpoints have been discovered, implementations can configure local Endpoints accordingly to establish communication.

The DDS specification equally relies on the use of a discovery mechanism to establish communication between matched DataWriters and DataReaders. DDS implementations must automatically discover the presence of remote entities, both when they join and leave the network. This discovery information is made accessible to the user through DDS built-in topics.

The RTPS discovery protocol defined in this Module provides the required discovery mechanism for DDS.

8.5.1 Overview

The RTPS specification splits up the discovery protocol into two independent protocols:

1. Participant Discovery Protocol
2. Endpoint Discovery Protocol

A Participant Discovery Protocol (PDP) specifies how Participants discover each other in the network. Once two Participants have discovered each other, they exchange information on the Endpoints they contain using an
**Endpoint** Discovery Protocol (EDP). Apart from this causality relationship, both protocols can be considered independent.

Implementations may choose to support multiple PDPs and EDPs, possibly vendor-specific. As long as two Participants have at least one PDP and EDP in common, they can exchange the required discovery information. For the purpose of interoperability, all RTPS implementations must provide at least the following discovery protocols:

1. Simple Participant Discovery Protocol (SPDP)
2. Simple Endpoint Discovery Protocol (SEDP)

Both are basic discovery protocols that suffice for small to medium scale networks. Additional PDPs and EDPs that are geared towards larger networks may be added to future versions of the specification.

Finally, the role of a discovery protocol is to provide information on discovered remote *Endpoints*. How this information is used by a *Participant* to configure its local *Endpoints* depends on the actual implementation of the RTPS protocol and is not part of the discovery protocol specification. For example, for the reference implementations introduced in 8.4.7, the information obtained on the remote *Endpoints* allows the implementation to configure:

- The RTPS *ReaderLocator* objects that are associated with each RTPS *StatelessWriter*.
- The RTPS *ReaderProxy* objects associated with each RTPS *StatefulWriter*.
- The RTPS *WriterProxy* objects associated with each RTPS *StatefulReader*.

The Discovery Module is organized as follows:

- The SPDP and SEDP rely on pre-defined RTPS built-in Writer and Reader Endpoints to exchange discovery information. 8.5.2 introduces these RTPS built-in Endpoints.
- The SPDP is discussed in 8.5.3.
- The SEDP is discussed in 8.5.4.

### 8.5.2 RTPS Built-in Discovery Endpoints

The DDS specification specifies that discovery takes place using “built-in” DDS *DataReaders* and *DataWriters* with pre-defined Topics and QoS.

There are four pre-defined built-in Topics: “DCPSParticipant,” “DCPSSubscription,” “DCPSPublication,” and “DCPSTopic.” The DataTypes associated with these Topics are also specified by the DDS specification and mainly contain Entity QoS values.

For each of the built-in Topics, there exists a corresponding DDS built-in DataWriter and DDS built-in DataReader. The built-in DataWriters are used to announce the presence and QoS of the local DDS Participant and the DDS Entities it contains (DataReaders, DataWriters and Topics) to the rest of the network. Likewise, the built-in DataReaders collect this information from remote Participants, which is then used by the DDS implementation to identify matching remote Entities. The built-in DataReaders act as regular DDS DataReaders and can also be accessed by the user through the DDS API.

The approach taken by the RTPS Simple Discovery Protocols (SPDP and SEDP) is analogous to the built-in Entity concept. RTPS maps each built-in DDS DataWriter or DataReader to an associated built-in RTPS *Endpoint*. These built-in Endpoints act as regular Writer and Reader Endpoints and provide the means to exchange the required discovery information between Participants using the regular RTPS protocol defined in the Behavior Module.

The SPDP, which concerns itself with how Participants discover each other, maps the DDS built-in Entities for the “DCPSParticipant” Topic. The SEDP, which specifies how to exchange discovery information on local Topics, DataWriters and DataReaders, maps the DDS built-in Entities for the “DCPSSubscription,” “DCPSPublication” and “DCPSTopic” Topics.
8.5.3 The Simple Participant Discovery Protocol

The purpose of a PDP is to discover the presence of other Participants on the network and their properties.

A Participant may support multiple PDPs, but for the purpose of interoperability, all implementations must support at least the Simple Participant Discovery Protocol.

8.5.3.1 General Approach

The RTPS Simple Participant Discovery Protocol (SPDP) uses a simple approach to announce and detect the presence of Participants in a domain.

For each Participant, the SPDP creates two RTPS built-in Endpoints: the SPDPbuiltinParticipantWriter and the SPDPbuiltinParticipantReader.

The SPDPbuiltinParticipantWriter is an RTPS Best-Effort StatelessWriter. The HistoryCache of the SPDPbuiltinParticipantWriter contains a single data-object of type SPDPdiscoveredParticipantData. The value of this data-object is set from the attributes in the Participant. If the attributes change, the data-object is replaced.

The SPDPbuiltinParticipantWriter periodically sends this data-object to a pre-configured list of locators to announce the Participant’s presence on the network. This is achieved by periodically calling StatelessWriter::unsent_changes_reset, which causes the StatelessWriter to resend all changes present in its HistoryCache to all locators. The periodic rate at which the SPDPbuiltinParticipantWriter sends out the SPDPdiscoveredParticipantData defaults to a PSM specified value. This period should be smaller than the leaseDuration specified in the SPDPdiscoveredParticipantData (see also 8.5.3.3.2).

The pre-configured list of locators may include both unicast and multicast locators. Port numbers are defined by each PSM. These locators simply represent possible remote Participants in the network, no Participant need actually be present. By sending the SPDPdiscoveredParticipantData periodically, Participants can join the network in any order.

The SPDPbuiltinParticipantReader receives the SPDPdiscoveredParticipantData announcements from the remote Participants. The contained information includes what Endpoint Discovery Protocols the remote Participant supports. The proper Endpoint Discovery Protocol is then used for exchanging Endpoint information with the remote Participant.

Implementations can minimize any start-up delays by sending an additional SPDPdiscoveredParticipantData in response to receiving this data-object from a previously unknown Participant, but this behavior is optional. Implementations may also enable the user to choose whether to automatically extend the pre-configured list of locators with new locators from newly discovered Participants. This enables asymmetric locator lists. These last two features are optional and not required for the purpose of interoperability.

8.5.3.2 SPDPdiscoveredParticipantData

The SPDPdiscoveredParticipantData defines the data exchanged as part of the SPDP.

Figure 8.27 illustrates the contents of the SPDPdiscoveredParticipantData. As shown in the figure, the SPDPdiscoveredParticipantData specializes the ParticipantProxy and therefore includes all the information necessary to configure a discovered Participant. The SPDPdiscoveredParticipantData also specializes the DDS-defined DDS::ParticipantBuiltinTopicData providing the information the corresponding DDS built-in DataReader needs.
The attributes of the **SPDPdiscoveredParticipantData** and their interpretation are described in Table 8.78.

**Table 8.78 - RTPS SPDPdiscoveredParticipantData attributes**

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>domainId</td>
<td>DomainId_t</td>
<td>Identifies the DDS domainId of the associated DDS DomainParticipant.</td>
</tr>
<tr>
<td>domainTag</td>
<td>string</td>
<td>Identifies the DDS domainTag of the associated DDS DomainParticipant.</td>
</tr>
<tr>
<td>protocolVersion</td>
<td>ProtocolVersion_t</td>
<td>Identifies the RTPS protocol version used by the Participant.</td>
</tr>
<tr>
<td>guidPrefix</td>
<td>GuidPrefix_t</td>
<td>The common GuidPrefix_t of the Participant and all the Endpoints contained within the Participant.</td>
</tr>
<tr>
<td>vendorId</td>
<td>VendorId_t</td>
<td>Identifies the vendor of the DDS middleware that contains the Participant.</td>
</tr>
<tr>
<td>expectsInlineQos</td>
<td>bool</td>
<td>Describes whether the Readers within the Participant expect that the QoS values that apply to each data modification are encapsulated included with each Data.</td>
</tr>
<tr>
<td>metatrafficUnicastLocatorList</td>
<td>Locator_t[*]</td>
<td>List of unicast locators (transport, address, port combinations) that can be used to send messages to the built-in Endpoints contained in the Participant.</td>
</tr>
<tr>
<td>metatrafficMulticastLocatorList</td>
<td>Locator_t[*]</td>
<td>List of multicast locators (transport, address, port combinations) that can be used to send messages to the built-in Endpoints contained in the Participant.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>defaultUnicastLocatorList</td>
<td>Default list of unicast locators (transport, address, port combinations) that can be used to send messages to the user-defined Endpoints contained in the Participant. These are the unicast locators that will be used in case the Endpoint does not specify its own set of Locators, so at least one Locator must be present.</td>
<td></td>
</tr>
<tr>
<td>defaultMulticastLocatorList</td>
<td>Default list of multicast locators (transport, address, port combinations) that can be used to send messages to the user-defined Endpoints contained in the Participant. These are the multicast locators that will be used in case the Endpoint does not specify its own set of Locators.</td>
<td></td>
</tr>
<tr>
<td>availableBuiltinEndpoints</td>
<td>All Participants must support the SEDP. This attribute identifies the kinds of built-in SEDP Endpoints that are available in the Participant. This allows a Participant to indicate that it only contains a subset of the possible built-in Endpoints. See also 8.5.3.2.1 and 9.3.2.12.</td>
<td></td>
</tr>
<tr>
<td>leaseDuration</td>
<td>How long a Participant should be considered alive every time an announcement is received from the Participant. If a Participant fails to send another announcement within this time period, the Participant can be considered gone. In that case, any resources associated to the Participant and its Endpoints can be freed.</td>
<td></td>
</tr>
<tr>
<td>manualLivelinessCount</td>
<td>Used to implement MANUAL_BY_PARTICIPANT liveliness QoS. When liveliness is asserted, the manualLivelinessCount is incremented and a new SPDPdiscoveredParticipantData is sent.</td>
<td></td>
</tr>
<tr>
<td>builtinEndpointQos</td>
<td>Provides additional information on the QoS of the built-in Endpoints supported by the Participant.</td>
<td></td>
</tr>
</tbody>
</table>

As mentioned in 8.5.3.1, the SPDPdiscoveredParticipantData lists the Endpoint Discovery Protocols supported by the Participant. The attributes shown in Table 8.78 only reflect the mandatory SEDP. There are currently no other Endpoint Discovery Protocols defined by the RTPS specification. In order to extend SPDPdiscoveredParticipantData to include additional EDPs, the standard RTPS extension mechanisms can be used. Please refer to 9.6.3 for additional information.

8.5.3.2.1 The availableBuiltinEndpoints attribute

The SPDPdiscoveredParticipantData provides information about the builtin endpoints supported by the Participant. This information is contained in the availableBuiltinEndpoints attribute.

The builtin endpoints that may be announced in the availableBuiltinEndpoints include: PUBLICATIONS_DETECTOR, PUBLICATIONS_ANNOUNCER, SUBSCRIPTIONS_DETECTOR, SUBSCRIPTIONS_ANNOUNCER, TOPICS_DETECTOR, TOPICS_ANNOUNCER, PARTICIPANT_MESSAGE_READER, and PARTICIPANT_MESSAGE_WRITER.

The availableBuiltinEndpoints may also announce builtin endpoints defined in other DDS specifications. See 9.3.2.12.

8.5.3.3 The built-in Endpoints used by the Simple Participant Discovery Protocol

Figure 8.28 illustrates the built-in Endpoints introduced by the Simple Participant Discovery Protocol.
The Protocol reserves the following values of the `EntityId_t` for the SPDP built-in Endpoints:

- ENTITYID_SPDP_BUILTIN_PARTICIPANT_WRITER
- ENTITYID_SPDP_BUILTIN_PARTICIPANT_READER

### 8.5.3.3.1 SPDPbuiltinParticipantWriter

The relevant attribute values for configuring the `SPDPbuiltinParticipantWriter` are shown in Table 8.79.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>unicastLocatorList</td>
<td>Locator_t[*]</td>
<td><code>&lt;auto-detected&gt;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transport-kinds and addresses are either auto-detected or configured by the application. Ports are a parameter to the SPDP initialization or else are set to a PSM-specified value that depends on the domainId.</td>
</tr>
<tr>
<td>multicastLocatorList</td>
<td>Locator_t[*]</td>
<td><code>&lt;parameter to the SPDP initialization&gt;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Defaults to a PSM-specified value.</td>
</tr>
<tr>
<td>reliabilityLevel</td>
<td>ReliabilityKind_t</td>
<td>BEST_EFFORT</td>
</tr>
<tr>
<td>topicKind</td>
<td>TopicKind_t</td>
<td>WITH_KEY</td>
</tr>
<tr>
<td>resendPeriod</td>
<td>Duration_t</td>
<td><code>&lt;parameter to the SPDP initialization&gt;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Defaults to a PSM-specified value.</td>
</tr>
<tr>
<td>readerLocators</td>
<td>ReaderLocator[*]</td>
<td><code>&lt;parameter to the SPDP initialization&gt;</code></td>
</tr>
</tbody>
</table>

### 8.5.3.3.2 SPDPbuiltinParticipantReader

The `SPDPbuiltinParticipantReader` is configured with the attribute values shown in Table 8.80.
Table 8.80 - Attributes of the RTPS StatelessReader used by the SPDP

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>unicastLocatorList</td>
<td>Locator_t[*]</td>
<td>&lt;auto-detected&gt; Transport-kinds and addresses are either auto-detected or configured by the application. Ports are a parameter to the SPDP initialization or else are set to a PSM-specified value that depends on the domainId.</td>
</tr>
<tr>
<td>multicastLocatorList</td>
<td>Locator_t[*]</td>
<td>&lt;parameter to the SPDP initialization&gt;. Defaults to a PSM-specified value.</td>
</tr>
<tr>
<td>reliabilityLevel</td>
<td>ReliabilityKind_t</td>
<td>BEST_EFFORT</td>
</tr>
<tr>
<td>topicKind</td>
<td>TopicKind_t</td>
<td>WITH_KEY</td>
</tr>
</tbody>
</table>

The HistoryCache of the SPDPbuiltinParticipantReader contains information on all active discovered participants; the key used to identify each data-object corresponds to the Participant GUID.

Each time information on a participant is received by the SPDPbuiltinParticipantReader, the SPDP examines the HistoryCache looking for an entry with a key that matches the Participant GUID. If an entry with a matching key is not there, a new entry is added keyed by the GUID of the Participant.

Periodically, the SPDP examines the SPDPbuiltinParticipantReader HistoryCache looking for stale entries defined as those that have not been refreshed for a period longer than their specified leaseDuration. Stale entries are removed.

8.5.3.4 Logical ports used by the Simple Participant Discovery Protocol

As mentioned above, each SPDPbuiltinParticipantWriter uses a pre-configured list of locators to announce a Participant’s presence on the network.

In order to enable plug-and-play interoperability, the pre-configured list of locators must use the following well-known logical ports:

Table 8.81 - Logical ports used by the Simple Participant Discovery Protocol

<table>
<thead>
<tr>
<th>Port</th>
<th>Locators configured using this port</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPDP_WELL_KNOWN_UNICAST_PORT</td>
<td>entries in SPDPbuiltinParticipantReader.unicastLocatorList, unicast entries in SPDPbuiltinParticipantWriter.readerLocators</td>
</tr>
<tr>
<td>SPDP_WELL_KNOWN_MULTICAST_PORT</td>
<td>entries in SPDPbuiltinParticipantReader.multicastLocatorList, multicast entries in SPDPbuiltinParticipantWriter.readerLocators</td>
</tr>
</tbody>
</table>

The actual value for the logical ports is defined by the PSM.
8.5.4 The Simple Endpoint Discovery Protocol

An Endpoint Discovery Protocol defines the required information exchange between two Participants in order to discover each other’s Writer and Reader Endpoints.

A Participant may support multiple EDPs, but for the purpose of interoperability, all implementations must support at least the Simple Endpoint Discovery Protocol.

8.5.4.1 General Approach

Similar to the SPDP, the Simple Endpoint Discovery Protocol uses pre-defined built-in Endpoints. The use of pre-defined built-in Endpoints means that once a Participant knows of the presence of another Participant, it can assume the presence of the built-in Endpoints made available by the remote participant and establish the association with the locally-matching built-in Endpoints.

The protocol used to communicate between built-in Endpoints is the same as used for application-defined Endpoints. Therefore, by reading the built-in Reader Endpoints, the protocol virtual machine can discover the presence and QoS of the DDS Entities that belong to any remote Participants. Similarly, by writing the built-in Writer Endpoints a Participant can inform the other Participants of the existence and QoS of local DDS Entities.

The use of built-in topics in the SEDP therefore reduces the scope of the overall discovery protocol to the determination of which Participants are present in the system and the attribute values for the ReaderProxy and WriterProxy objects that correspond to the built-in Endpoints of these Participants. Once that is known, everything else results from the application of the RTPS protocol to the communication between the built-in RTPS Readers and Writers.

8.5.4.2 The built-in Endpoints used by the Simple Endpoint Discovery Protocol

The SEDP maps the DDS built-in Entities for the “DCPSSubscription,” “DCPSPublication,” and “DCPSTopic” Topics. According to the DDS specification, the reliability QoS for these built-in Entities is set to ‘reliable.’ The SEDP therefore maps each corresponding built-in DDS DataWriter or DataReader into corresponding reliable RTPS Writer and Reader Endpoints.

For example, as illustrated in Figure 8.29, the DDS built-in DataWriters for the “DCPSSubscription,” “DCPSPublication,” and “DCPSTopic” Topics can be mapped to reliable RTPS StatefulWriters and the corresponding DDS built-in DataReaders to reliable RTPS StatefulReaders. Actual implementations need not use the stateful reference implementation. For the purpose of interoperability, it is sufficient that an implementation provides the required built-in Endpoints and reliable communication that satisfies the general requirements listed in 8.4.2.

![Diagram](image)

Figure 8.29 - Example mapping of the DDS Built-in Entities to corresponding RTPS built-in Endpoints

The RTPS Protocol reserves the following values of the EntityId_t for the built-in Endpoints:
The actual value for the reserved "EntityId_t" is defined by each PSM.

8.5.4.3 Built-in Endpoints required by the Simple Endpoint Discovery Protocol

Implementations are not required to provide all built-in Endpoints.

As mentioned in the DDS specification, Topic propagation is optional. Therefore, it is not required to implement the SEDPbuiltinTopicsReader and SEDPbuiltinTopicsWriter built-in Endpoints and for the purpose of interoperability, implementations should not rely on their presence in remote Participants.

As far as the remaining built-in Endpoints are concerned, a Participant is only required to provide the built-in Endpoints required for matching up local and remote Endpoints. For example, if a DDS Participant will only contain DDS DataWriters, the only required RTPS built-in Endpoints are the SEDPbuiltinPublicationsWriter and the SEDPbuiltinSubscriptionsReader. The SEDPbuiltinPublicationsReader and the SEDPbuiltinSubscriptionsWriter built-in Endpoints serve no purpose in this case.

The SPDP specifies how a Participant informs other Participants about what built-in Endpoints it has available. This is discussed in 8.5.3.2.

8.5.4.4 Data Types associated with built-in Endpoints used by the Simple Endpoint Discovery Protocol

Each RTPS Endpoint has a HistoryCache that stores changes to the data-objects associated with the Endpoint. This also applies to the RTPS built-in Endpoints. Therefore, each RTPS built-in Endpoint depends on some DataType that represents the logical contents of the data written into its HistoryCache.

Figure 8.30 defines the DiscoveredWriterData, DiscoveredReaderData, and DiscoveredTopicData DataTypes associated with the RTPS built-in Endpoints for the “DCPSPublication,” “DCPSSubscription,” and “DCPSTopic” Topics. The DataType associated with the “DCPSParticipant” Topic is defined in 8.5.3.2.

The DataType associated with each RTPS built-in Endpoint contains all the information specified by DDS for the corresponding built-in DDS Entity. For this reason, DiscoveredReaderData extends the DDS-defined DDS::SubscriptionBuiltinTopicData, DiscoveredWriterData extends DDS::PublicationBuiltinTopicData, and DiscoveredTopicData extends DDS::TopicBuiltinTopicData.

In addition to the data needed by the associated built-in DDS Entities, the “Discovered” DataTypes also include all the information that may be needed by an implementation of the protocol to configure the RTPS Endpoints. This information is contained in the RTPS ReaderProxy and WriterProxy.
Figure 8.30 - Data types associated with built-in Endpoints used by the Simple Endpoint Discovery Protocol

An implementation of the protocol need not necessarily send all information contained in the DataTypes. If any information is not present, the implementation can assume the default values, as defined by the PSM. The PSM also defines how the discovery information is represented on the wire.

The RTPS built-in Endpoints used by the SEDP and their associated DataTypes are shown in Figure 8.31.
The contents of the **HistoryCache** for each built-in Endpoint can be described in terms of the following aspects: DataType, Cardinality, Data-object insertion, Data-object modification, and Data-object deletion.

- **DataType.** The type of the data stored in the cache. This is partly defined by the DDS specification.
- **Cardinality.** The number of different data-objects (each with a different key) that can potentially be stored in the cache.
- **Data-object insertion.** Conditions under which a new data-object is inserted into the cache.
- **Data-object modification.** Conditions under which the value of an existing data-object is modified.
- **Data-object deletion.** Conditions under which an existing data-object is removed from the cache.

It is illustrative to describe the **HistoryCache** for each of the built-in Endpoints.

### 8.5.4.4.1 SEDPbuiltinPublicationsWriter and SEDPbuiltinPublicationsReader

Table 8.82 describes the **HistoryCache** for the **SEDPbuiltinPublicationsWriter** and **SEDPbuiltinPublicationsReader**.

<table>
<thead>
<tr>
<th>aspect</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DataType</strong></td>
<td>DiscoveredWriterData</td>
</tr>
<tr>
<td><strong>Cardinality</strong></td>
<td>The number of DataWriters contained by the DomainParticipant. There is a one-to-one correspondence between each DataWriter in the participant and a data-object that describes the DataWriter stored in the WriterHistoryCache for the SEDPbuiltinPublicationsWriter.</td>
</tr>
<tr>
<td><strong>Data-Object insertion</strong></td>
<td>Each time a DataWriter is created in the DomainParticipant.</td>
</tr>
<tr>
<td><strong>Data-Object modification</strong></td>
<td>Each time the QoS of an existing DataWriter is modified.</td>
</tr>
<tr>
<td><strong>Data-Object deletion</strong></td>
<td>Each time an existing DataWriter belonging to the DomainParticipant is deleted.</td>
</tr>
</tbody>
</table>

### 8.5.4.4.2 SEDPbuiltinSubscriptionsWriter and SEDPbuiltinSubscriptionsReader

Table 8.83 describes the HistoryCache for the SEDPbuiltinSubscriptionsWriter and SEDPbuiltinSubscriptionsReader.
Table 8.83 - Contents of the HistoryCache for the SEDPbuiltinSubscriptionsWriter and SEDPbuiltinSubscriptionsReader

<table>
<thead>
<tr>
<th>aspect</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataType</td>
<td>DiscoveredReaderData</td>
</tr>
<tr>
<td>Cardinality</td>
<td>The number of DataReaders contained by the DomainParticipant. There is a one-to-one correspondence between each DataReaders in the Participant and a data-object that describes the DataReaders stored in the WriterHistoryCache for the SEDPbuiltinSubscriptionsWriter.</td>
</tr>
<tr>
<td>Data-Object insertion</td>
<td>Each time a DataReader is created in the DomainParticipant.</td>
</tr>
<tr>
<td>Data-Object modification</td>
<td>Each time the QoS of an existing DataReader is modified.</td>
</tr>
<tr>
<td>Data-Object deletion</td>
<td>Each time an existing DataReader belonging to the DomainParticipant is deleted.</td>
</tr>
</tbody>
</table>

8.5.4.4.3 SEDPbuiltinTopicsWriter and SEDPbuiltinTopicsReader

Table 8.84 describes the HistoryCache for the SEDPbuiltinTopicsWriter and builtinTopicsReader.

Table 8.84 - Contents of the HistoryCache for the SEDPbuiltinTopicsWriter and SEDPbuiltinTopicsReader

<table>
<thead>
<tr>
<th>aspect</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataType</td>
<td>DiscoveredTopicData</td>
</tr>
<tr>
<td>Cardinality</td>
<td>The number of Topics created by the DomainParticipant. There is a one-to-one correspondence between each Topic created by the DomainParticipant and a data-object that describes the Topic stored in the WriterHistoryCache for the builtinTopicsWriter.</td>
</tr>
<tr>
<td>Data-Object insertion</td>
<td>Each time a Topic is created in the DomainParticipant.</td>
</tr>
<tr>
<td>Data-Object modification</td>
<td>Each time the QoS of an existing Topic is modified.</td>
</tr>
<tr>
<td>Data-Object deletion</td>
<td>Each time an existing Topic belonging to the DomainParticipant is deleted.</td>
</tr>
</tbody>
</table>

8.5.5 Interaction with the RTPS virtual machine

To further illustrate the SPDP and SEDP, this specification describes how the information provided by the SPDP can be used to configure the SEDP built-in Endpoints in the RTPS virtual machine.

8.5.5.1 Discovery of a new remote Participant

Using the SPDPbuiltinParticipantReader, a local Participant ‘local_participant’ discovers the existence of another Participant described by the DiscoveredParticipantData participant_data. The discovered Participant uses the SEDP.

The pseudo code below configures the local SEDP built-in Endpoints within local_participant to communicate with the corresponding SEDP built-in Endpoints in the discovered Participant.

Note that how the Endpoints are configured depends on the implementation of the protocol. For the stateful reference implementation, this operation performs the following logical steps:

```
// Check that the domainId of the discovered participant equals the local one.
// If it is not equal then there the local endpoints are not configured to
// communicate with the discovered participant.
IF ( participant_data.domainId != local_participant.domainId ) THEN
    RETURN;
```
// Check that the domainTag of the discovered participant equals the local one. // If it is not equal then there the local endpoints are not configured to communicate with the discovered participant.
IF ( !STRING_EQUAL(participant_data.domainTag, localParticipant.domainTag) ) THEN
  RETURN;
ENDIF

IF ( PUBLICATIONS_DETECTOR IS_IN participant_data.availableEndpoints ) THEN
  guid = <participant_data.guidPrefix, ENTITYID_SEDP_BUILTIN_PUBLICATIONS_DETECTOR>; writer = localParticipant.SEDPbuiltinPublicationsWriter;
  proxy = new ReaderProxy( guid, participant_data.metatrafficUnicastLocatorList, participant_data.metatrafficMulticastLocatorList);
  writer.matched_reader_add(proxy);
ENDIF

IF ( PUBLICATIONS_ANNOUNCER IS_IN participant_data.availableEndpoints ) THEN
  guid = <participant_data.guidPrefix, ENTITYID_SEDP_BUILTIN_PUBLICATIONS_ANNOUNCER>
  reader = localParticipant.SEDPbuiltinPublicationsReader;
  proxy = new WriterProxy( guid, participant_data.metatrafficUnicastLocatorList, participant_data.metatrafficMulticastLocatorList);
  reader.matched_writer_add(proxy);
ENDIF

IF ( SUBSCRIPTIONS_DETECTOR IS_IN participant_data.availableEndpoints ) THEN
  guid = <participant_data.guidPrefix, ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_DETECTOR>; writer = localParticipant.SEDPbuiltinSubscriptionsWriter;
  proxy = new ReaderProxy( guid, participant_data.metatrafficUnicastLocatorList, participant_data.metatrafficMulticastLocatorList);
  writer.matched_reader_add(proxy);
ENDIF

IF ( SUBSCRIPTIONS_ANNOUNCER IS_IN participant_data.availableEndpoints ) THEN
  guid = <participant_data.guidPrefix, ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_ANNOUNCER>
  reader = localParticipant.SEDPbuiltinSubscriptionsReader;
  proxy = new WriterProxy( guid, participant_data.metatrafficUnicastLocatorList, participant_data.metatrafficMulticastLocatorList);
  reader.matched_writer_add(proxy);
ENDIF

IF ( TOPICS_DETECTOR IS_IN participant_data.availableEndpoints ) THEN
  guid = <participant_data.guidPrefix, ENTITYID_SEDP_BUILTIN_TOPICS_DETECTOR>; writer = localParticipant.SEDPbuiltinTopicsWriter;
  proxy = new ReaderProxy( guid, participant_data.metatrafficUnicastLocatorList, participant_data.metatrafficMulticastLocatorList);
  writer.matched_reader_add(proxy);
ENDIF

IF ( TOPICS_ANNOUNCER IS_IN participant_data.availableEndpoints ) THEN
  guid = <participant_data.guidPrefix, ENTITYID_SEDP_BUILTIN_TOPICS_ANNOUNCER>
  reader = localParticipant.SEDPbuiltinTopicsReader;
  proxy = new WriterProxy( guid,
8.5.5.2 Removal of a previously discovered Participant

Based on the remote Participant’s leaseDuration, a local Participant ‘local_participant’ concludes that a previously discovered Participant with GUID_t participant_guid is no longer present. The Participant ‘local_participant’ must reconfigure any local Endpoints that were communicating with Endpoints in the Participant identified by the GUID_t participant_guid.

For the stateful reference implementation, this operation performs the following logical steps:

guid = <participant_guid.guidPrefix,
   ENTITYID_SEDP_BUILTIN_PUBLICATIONS_DETECTOR>;
writer = local_participant.SEDPbuiltinPublicationsWriter;
proxy = writer.matched_reader_lookup(guid);
writer.matched_reader_remove(proxy);
guid = <participant_guid.guidPrefix,
   ENTITYID_SEDP_BUILTIN_PUBLICATIONS_ANNOUNCER>;
reader = local_participant.SEDPbuiltinPublicationsReader;
proxy = reader.matched_writer_lookup(guid);
reader.matched_writer_remove(proxy);
guid = <participant_guid.guidPrefix,
   ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_DETECTOR>;
writer = local_participant.SEDPbuiltinSubscriptionsWriter;
proxy = writer.matched_reader_lookup(guid);
writer.matched_reader_remove(proxy);
guid = <participant_guid.guidPrefix,
   ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_ANNOUNCER>;
reader = local_participant.SEDPbuiltinSubscriptionsReader;
proxy = reader.matched_writer_lookup(guid);
reader.matched_writer_remove(proxy);
guid = <participant_guid.guidPrefix,
   ENTITYID_SEDP_BUILTIN_TOPICS_DETECTOR>;
writer = local_participant.SEDPbuiltinTopicsWriter;
proxy = writer.matched_reader_lookup(guid);
writer.matched_reader_remove(proxy);
guid = <participant_guid.guidPrefix, ENTITYID_SEDP_BUILTIN_TOPICS_ANNOUNCER>;
reader = local_participant.SEDPbuiltinTopicsReader;
proxy = reader.matched_writer_lookup(guid);
reader.matched_writer_remove(proxy);

8.5.6 Supporting Alternative Discovery Protocols

The requirements on the Participant and Endpoint Discovery Protocols may vary depending on the deployment scenario. For example, a protocol optimized for speed and simplicity (such as a protocol that would be deployed in embedded devices on a LAN) may not scale well to large systems in a WAN environment.

For this reason, the RTPS specification allows implementations to support multiple PDPs and EDPs. There are many possible approaches to implementing a Discovery Protocol including the use of static discovery, file-based discovery, a central look-up service, etc. The only requirement imposed by RTPS for the purpose of interoperability is that all RTPS implementations support at least the SPDP and SEDP. It is expected that over time, a collection of interoperable Discovery Protocols will be developed to address specific deployment needs.

If an implementation supports multiple PDPs, each PDP may be initialized differently and discover a different set of remote Participants. Remote Participants using a different vendor’s RTPS implementation must be contacted using at least the SPDP to ensure interoperability. There is no such requirement when the remote Participant uses the same RTPS implementation.
Even when the SPDP is used by all Participants, remote Participants may still use different EDPs. Which EDPs a Participant supports is included in the information exchanged by the SPDP. All Participants must support at least the SEDP, so they always have at least one EDP in common. However, if two Participants both support another EDP, this alternative protocol can be used instead. In that case, there is no need to create the SEDP built-in Endpoints, or if they already exist, no need to configure them to match the new remote Participant. This approach enables a vendor to customize the EDP if desired without compromising interoperability.

8.6 Versioning and Extensibility

Implementations of this version of the RTPS protocol should be able to process RTPS Messages not only with the same major version but possibly higher minor versions.

8.6.1 Allowed Extensions within this major Version

Within this major version, future minor versions of the protocol can augment the protocol in the following ways:

- Additional Submessages with other submessageIds can be introduced and used anywhere in an RTPS Message. An implementation should skip over unknown Submessages using the submessageLength field in the SubmessageHeader.
- Additional fields can be added to the end of a Submessage that was already defined in the current minor version. An implementation should skip over additional fields using the submessageLength field in the SubmessageHeader.
- Additional built-in Endpoints with new IDs can be added. An implementation should ignore any unknown built-in Endpoints. Additional parameters with new parameterIds can be added. An implementation should ignore any unknown parameters.

All such changes require an increase of the minor version number.

8.6.2 What cannot change within this major Version

The following items cannot be changed within the same major version:

- A Submessage cannot be deleted.
- A Submessage cannot be modified except as described in 8.6.1.
- The meaning of submessageIds cannot be modified.

All such changes require an increase in the major version number.

8.7 Implementing DDS QoS and advanced DDS features using RTPS

The RTPS protocol and its extension mechanisms provide the core functionality required to implement DDS. This sub clause defines how to use RTPS to implement the DDS QoS parameters.

In addition, this sub clause defines the RTPS protocol extensions required for implementing the following advanced DDS features:

- Content-filtered Topics, see 8.7.3
- Instance State Changes 8.7.4
- Group Ordered Access, see 8.7.5
- Coherent Sets, see 8.7.6

All extensions are based on the standard extension mechanisms provided by RTPS.

This sub clause forms a normative part of the specification for the purpose of interoperability.
8.7.1 Adding in-line Parameters to Data Submessages

Data and DataFrag Submessages optionally contain a ParameterList SubmessageElement for storing in-line QoS parameters and other information.

In case a Reader does not keep a list of matching remote Writers or the QoS parameters they were configured with (i.e., is a stateless Reader), a Data Submessage with in-line QoS parameters contains all the information needed to enable the Reader to apply all Writer-specific QoS parameters.

A stateless Reader’s need for receiving in-line QoS to get information on remote Writers is the justification for requiring a Writer to send in-line QoS if the Reader requests them (8.4.2.2.2).

For immutable QoS, all RxO QoS are sent in-line to allow a stateless Reader to reject samples in case of incompatible QoS. Mutable QoS relevant to the Reader are sent in-line so they may take effect immediately, regardless of the amount of state kept on the Reader. Note that a stateful Reader has the option of relying on its cached information of remote Writers rather than the received in-line QoS.

A stateless Reader uses the discovery protocol to announce to remote Writers that it expects to receive QoS parameters in-line, as discussed in the Discovery Module (8.5). If in-line QoS parameters are expected, implementations must also include the topic name as an in-line parameter. This ensures that on the receiving side, the Submessage can be passed to all Readers for that topic, including the stateless Readers.

Independent of whether Readers expect in-line QoS parameters, a Data Submessage may also contain in-line parameters related to coherent sets and content-filtered topics. This is described in more detail in the sub clauses that follow.

For improved performance, stateful implementations may ignore in-line QoS and instead rely solely on cached values obtained through Discovery. Note that not parsing in-line QoS may delay the point in time when a new QoS takes effect, as it first must be propagated through Discovery.

8.7.2 DDS QoS Parameters

Table 8.85 provides an overview of which QoS parameters affect the RTPS wire protocol and which can appear as in-line QoS. The parameters that affect the wire protocol are discussed in more detail in the subsub clauses below.

Table 8.85 - Implementing DDS QoS Parameters using the RTPS Wire Protocol

<table>
<thead>
<tr>
<th>QoS</th>
<th>Effect on RTPS Protocol</th>
<th>May appear as in-line QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER_DATA</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>TOPIC_DATA</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>GROUP_DATA</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>DURABILITY</td>
<td>See 8.7.2.2.1</td>
<td>Yes</td>
</tr>
<tr>
<td>DURABILITY_SERVICE</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>PRESENTATION</td>
<td>See 8.7.2.2.2</td>
<td>Yes</td>
</tr>
<tr>
<td>DEADLINE</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>LATENCY_BUDGET</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>OWNERSHIP</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>OWNERSHIP_STRENGTH</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>LIVELINESS</td>
<td>See 8.7.2.2.3</td>
<td>Yes</td>
</tr>
</tbody>
</table>
8.7.2.1 In-line DDS QoS Parameters

Table 8.85 lists the standard DDS QoS parameters that may appear in-line.

If a Reader expects to receive in-line QoS parameters and any of these QoS parameters are missing, it will assume the default value for that QoS parameter, where the default is defined by DDS.

In-line parameters are added to data submessages to make them self-describing. In order to achieve self-describing messages, not only the parameters defined in Table 8.85 have to be sent with the submessage, but also a parameter TOPIC_NAME. This parameter contains the name of the topic that the submessage belongs to.

8.7.2.2 DDS QoS Parameters that affect the wire protocol

8.7.2.2.1 DURABILITY

While volatile and transient-local durability do not affect the RTPS protocol, support for transient and persistent durability may. This is not covered in the current version of the specification.

8.7.2.2.2 PRESENTATION

Sub clause 8.7.5 defines how to implement the GROUP ordered access policy of the PRESENTATION QoS.

Sub clause 8.7.6 defines how to implement the coherent access policy of the PRESENTATION QoS. The other aspects of this QoS do not affect the RTPS protocol.

8.7.2.2.3 LIVELINESS

Implementations must follow the approaches below:

- DDS_AUTOMATIC_LIVELINESS_QOS: liveliness is maintained through the BuiltinParticipantMessageWriter. For a given Participant, in order to maintain the liveliness of its Writer Entities with LIVELINESS QoS set to AUTOMATIC, implementations must refresh the Participant's liveliness (i.e., send the ParticipantMessageData, see (8.4.13.5) at a rate faster than the smallest lease duration among the Writers.

- DDS_MANUAL_BY_PARTICIPANT_LIVELINESS_QOS: liveliness is maintained through the BuiltinParticipantMessageWriter. If the Participant has any MANUAL_BY_PARTICIPANT Writers, implementations must check periodically to see if write(), assert_liveliness(), dispose(), or unregister_instance() was called for any of them. The period for this check equals the smallest lease duration among the Writers. If any of the operations were called, implementations must refresh the Participant's liveliness (i.e., send the ParticipantMessageData, see 8.4.13.5).
• DDS_MANUAL_BY_TOPIC_LIVENESS_QOS: liveness is maintained by sending data or an explicit Heartbeat message with liveness flag set. The standard RTPS Messages that result from calling write(), dispose(), or unregister_instance() on a Writer Entity suffice to assert the liveness of a Writer with LIVENESS QoS set to MANUAL_BY_TOPIC. When assert_liveliness() is called, the Writer must send a Heartbeat Message with final flag and liveness flag set.

8.7.2.4 TIME_BASED_FILTER

Implementations may optimize bandwidth usage by applying a time-based filter on the Writer side. That way, data that would be dropped on the Reader side is never sent.

When one or more data updates are filtered out on the Writer side, implementations must send a Gap Submessage instead, indicating which samples were filtered out. This Submessage must be sent before the next update and notifies the Reader the missing updates were filtered out and not simply lost.

8.7.2.5 RELIABILITY

Implementations must meet the reliable RTPS protocol requirements for interoperability, defined in 8.4.2.

8.7.2.6 DESTINATION_ORDER

In order to implement the DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS policy, implementations must include an InfoTimestamp Submessage with every update from a Writer.

8.7.2.7 WRITER_DATA_LIFECYCLE

If autodispose_unregisterd_instances is enabled, Data Messages that unregister an instance must also dispose it. This restricts the allowable values of the DisposedFlag and UnregisteredFlag flags.

8.7.3 Content-filtered Topics

Content-filtered topics make it possible for a DDS DataReader to request the middleware to filter out data samples based on their contents.

When filtering on the Reader side only, samples which do not pass the filter are simply dropped by the middleware. In this case, no further extensions to RTPS are needed.

In many cases, implementations will benefit from filtering on the Writer side, in addition to filtering on the Reader side. When filtering on the Writer side, a sample that does not pass a Reader side filter may sometimes not be sent to that Reader. This conserves bandwidth.

In order to support Writer side filtering, standard RTPS extension mechanisms are used to:

• Include Reader filter information during the Endpoint discovery phase.
• Include filter results with each data sample.

The Writer may indicate to a Reader that a Sample has been filtered due to the application of the reader-specified content filter by sending a directed Data message that includes only the key information (DataFlag=0), indicating in the Inline Qos that the instance state is ALIVE_FILTERED. See 8.7.3.2. The Reader may use this information to transition the specified instance to InstanceState ALIVE_FILTERED.

The Writer may indicate to a Reader that it has applied a set of filters to a Sample and the corresponding result by including the ContentFilteredInfo_t into the Data message, see 8.7.3.3. Readers can use ContentFilteredInfo_t to determine whether their filter has been already applied by the Writer and avoid having to apply the filter again.

Alternatively, the Writer may not send a Data message at all. This is only allowed if the previous sample for that Instance was already filtered for that Reader, see 8.7.4.
8.7.3.1 Exchanging filter information using the built-in Endpoints

Content-filtered topics are defined on the Reader side. In order to implement Writer side filtering, information on the filter used by a given Reader must be propagated to matching remote Writers. This requires extending the data type associated with RTPS built-in Endpoints.

As illustrated in Figure 8.31, the data types associated with RTPS built-in Endpoints extend the DDS built-in topic data types, which include all relevant QoS. Since DDS does not define content-filtered topics as a Reader QoS policy (instead, DDS defines separate Content-filtered Topics), RTPS adds an additional `ContentFilterProperty_t` field to DiscoveredReaderData, defined in Table 8.86.

Table 8.86 - Content filter property

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>contentFilteredTopicName</td>
<td>string</td>
<td>Name of the Content-filtered Topic associated with the Reader. Must have non-zero length.</td>
</tr>
<tr>
<td>relatedTopicName</td>
<td>string</td>
<td>Name of the Topic related to the Content-filtered Topic. Must have non-zero length.</td>
</tr>
<tr>
<td>filterClassName</td>
<td>string</td>
<td>Identifies the filter class this filter belongs to. RTPS can support multiple filter classes (SQL, regular expressions, custom filters, etc). Must have non-zero length. RTPS predefines the following values: “DDSSQL” Default filter class name if none specified. Matches the SQL filter specified by DDS, which must be available in all implementations.</td>
</tr>
<tr>
<td>filterExpression</td>
<td>string</td>
<td>The actual filter expression. Must be a valid expression for the filter class specified using <code>filterClassName</code>. Must have non-zero length.</td>
</tr>
<tr>
<td>expressionParameters</td>
<td>stringSequence</td>
<td>Defines the value for each parameter in the filter expression. Can have zero length if the filter expression contains no parameters.</td>
</tr>
</tbody>
</table>

The `ContentFilterProperty_t` field provides all the required information to enable content filtering on the Writer side. For example, for the default DDSSQL filter class, a valid filter expression for a data type containing members a, b and c could be “(a < 5) AND (b == %0) AND (c >= %1)” with expression parameters “5” and “3.” In order for the Writer to apply the filter, it must have been configured to handle filters of the specified filter class. If not, the Writer will simply ignore the filter information and not filter any data samples.

DDS allows the user to modify the filter expression parameters at run-time. Each time the parameters are modified, the updated information is exchanged using the Endpoint discovery protocol. This is identical to updating a mutable QoS value.

8.7.3.2 Indicating to a Reader that a Sample has been filtered

There are situations when a Writer needs to communicate to a Reader that a sample was written but it does not pass the reader-specified Content Filter. When this happens, the Writer can use a Data submessage that does not contain a Data payload (DataFlag=0) and sets FilteredFlag=1, see 8.3.8.2.2.
8.7.3.3 Including in-line filter results with each data sample

In general, when applying filtering on the Writer side, a sample is not sent if it does not pass the remote Reader’s filter. In that case, the Data submessage is replaced by a Gap submessage. This ensures the sample is not considered ‘lost’ on the Reader side. This approach matches that of applying a time-based filter on the Writer side. The remainder of the discussion only refers to Data Submessages, but the same approach is followed for DataFrag Submessages.

In some cases, it may still be possible for a Reader to receive a sample that did not pass its filter, for example when sending data using multicast. Another use case is multiple Readers belonging to the same Participant. In that case, the Writer need only send a single RTPS message, destined to ENTITYID_UNKNOWN (see 8.4.15.5). Each Reader may use a different filter however, in which case the Writer needs to apply multiple filters before sending the sample.

In both use cases, two options exist:
1. The sample passes none of the filters for any of the remote Readers. In that case, the Data submessage is again replaced by a Gap submessage.
2. The sample passes some or all of the filters. In that case, the sample must still be sent and the writer must include information with the Data submessage on what filters were applied and the according result.

The inlineQos element of the Data submessage is used to include the necessary filter information. More specifically, a new parameter is added, containing the information shown in Table 8.87.

Table 8.87 - Content filter info associated with a data sample

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>filterResult</td>
<td>FilterResult_t</td>
<td>For each filter signature, the results indicate whether the sample passed the filter.</td>
</tr>
<tr>
<td>filterSignatures</td>
<td>FilterSignature_t[]</td>
<td>A list of filters that were applied to the sample.</td>
</tr>
</tbody>
</table>

A filter signature FilterSignature_t uniquely identifies a filter and is based on the filter properties listed in Table 8.86. How to represent and calculate a filter signature is defined by the PSM. Whether the sample passed the filters that were applied on the Writer side is encoded by the filterResult_t attribute, again defined by the PSM.

Note that a filter signature changes when the filter’s expression parameters change. Until it receives updated parameter values, a Writer side filter may be using outdated expression parameters, in which case the in-line filter signature will not match the signature expected by the Reader. As a result, the Reader will ignore the filter results and instead apply its local filter.

8.7.3.4 Requirements for Interoperability

Writer side filtering constitutes an optimization and is optional, so it is not required for interoperability. Samples will always be filtered on the Reader side if:
- The Writer side did not apply any filtering.
- The Writer side did not apply the filter expected by the Reader. As mentioned earlier, this may occur if the Writer has not yet been informed about updated filter parameters.
- The Reader side does not support Writer side filtering (and therefore ignores in-line filter information).

Likewise, Writers may not filter samples because:
• The implementation does not support Content-filtered Topics (in which case the filter properties of the Reader are ignored).

• The Reader's filter information was rejected (e.g., unrecognized filter class). If an implementation supports Content-filtered Topics, it must at least recognize the “DDSQL” filter class, as mandated by the DDS specification. For all other filter classes, both implementations must allow the user to register the same custom filter class.

• Other implementation-specific restrictions, such as a resource limit on the number of remote readers each writer is able to store filter information for.

Even if the **Writer** is performing writer-side filtering, the **Writer** must provide enough information for the **Reader** to correctly transition the instance state to ALIVE_FILTERED. This means that even if a Sample does not pass the reader filter, the **Writer** must still send a **Data** submessage unless it the previous sample for that Instance also did not pass the content filter. See §8.7.3.2.

This requirement effectively means that a **Writer** needs maintain state per Instance and per “content filtered” **Reader**. In this state it must remember whether the last sample written to that Instance passed the reader filter.

### 8.7.4 Changes in the Instance State

A DDS DataWriter may register data object instances (operation `register_instance`), update their value (operation `write`), dispose data-object instances (operation `dispose`), and unregister them (operation `unregister_instance`). When the value of an instance is updated, the new value may not pass the content filter specified by a subset of the DataReaders.

Each one of these operations may cause notifications to be dispatched to the matched DDS DataReaders. The DDS DataReader can determine the nature of the change by inspecting the **InstanceState instance_state** field in the **SampleInfo** that is returned on the DDS DataReader **read** or **take** call.

RTPS uses regular **Data** Submessages and the in-line QoS parameter extension mechanism to communicate instance state changes. The serialized information within the inline QoS contains the new **InstanceState**, that is, whether the instance has been registered, unregistered, or disposed. The actual details depend on the PSM (e.g., §9.6.4.4).

When RTPS sends a **Data** Submessage to communicate instance state changes it may include only the Key of the Data-Object within the SerializedPayload submessage element (see §8.3.8.2). This is because the Key is sufficient to uniquely identify the Data-Object instance to which the **InstanceState** change applies.

An implementation of RTPS is not required to propagate registration changes until the DDS DataWriter writes the first value for that Data-Object instance.

If a DataWriter updates the value of an instance (operation `write`), the updated value may not pass the content filter specified by one (or more) matched DataReaders. In this situation, there are two possibilities:

1. If the previous update to the instance passed the filter, then the Writer must send a Data Submessage that either includes the data value, or else indicates the InstanceState is ALIVE_FILTERED. See §9.6.4.5.

2. If the previous update to the instance did not pass the filter, then the Writer may omit sending the Data Submessage to the Reader.

The rules above ensure the Writer provides enough information for the Reader to transition the instance state to ALIVE_FILTERED.

If a DataWriter disposes an instance (operation `dispose`) or unregisters an instance (operation `unregister`), there are several possibilities which dictate whether the Writer must send a **Data** Submessage that indicates the **InstanceState** is NOT_ALIVE_DISPOSED or NOT_ALIVE_NO_WRITERS, respectively. This so called “dispose/unregister message” shall be sent if any of the following conditions is met:
1. The Reader does not have a Content Filter.
2. The Writer has previously sent a Data message to the Reader for that same instance.
3. The Reader has OWNERSHIP QosPolicy kind EXCLUSIVE and the Reader Filter is such that there could be some values for the Instance that pass the filter.

In all other cases, the “instance state change” message may be omitted as an optimization.

These conditions ensure that the Reader is able to determine consistently the ownership and InstanceState for the instance.

8.7.5 Group Ordered Access

The DDS Specification provides the functionality for CacheChanges made by DataWriter entities attached to the same Publisher object to be made available to subscribers in the same order they occur.

In order to support group ordered access, RTPS uses the in-line QoS parameter extension mechanism to include additional information with each CacheChange. The additional information denotes ordering within the scope of the Publisher, as well as the identity of the Writers belonging to the Publisher.

- PID_GROUP_SEQ_NUM to contain the group sequence number.
- PID_WRITER_GROUP_INFO to contain the WriterGroupInfo_t defined in Table 8.88.

Table 8.88 – Group Writer Info associated with a data sample

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>writerSet</td>
<td>GroupDigest_t</td>
<td>Identifies the set of Writer EntityIds that are announced in the DiscoveredWriterData that belonged to the Publisher at the time the sample was written.</td>
</tr>
</tbody>
</table>

When a Publisher is configured with access scope GROUP, all Data submessages and the first DataFrag submessage from any Writer within the Publisher are accompanied with a GROUP sequence number sent as part of the in-line QoS. The GROUP sequence number is a strictly monotonically increasing sequence number originating from the Publisher. Each time that a DataWriter attached to a Publisher makes a CacheChange (i.e., increments its own Writer sequence number), the GROUP sequence number is incremented.

A DataReader attached to a Subscriber configured with access scope GROUP first orders the samples from a remote Writer as it would in the cases where access scope GROUP is not set. Once a sample is ready to be committed to the DDS DataReader, it will not commit it. Instead, it will hand it off to a HistoryCache of the Subscriber where ordering across remote DataWriters belonging to the same Publisher occurs. A sample with GROUP sequence number GSN can be committed to the DDS DataReader from the Subscriber’s history cache if any of the following conditions apply:

- GSN-1 has been already been committed.
- It has been determined that none of the remote DataWriters that match reliable DataReaders have GSN-1. This condition is met when both of the following conditions apply:
  - The Subscriber has received a Heartbeat from one of the DataWriters with Heartbeat.currentGSN.value >= GSN and the Heartbeat.writerSet (and Heartbeat.secureWriterSet) matches the set of discovered DataWriters.
AND for every matched DataWriter belonging to the Publisher that matches a reliableDataReader, the DataWriter has:

- Either advanced past the GSN-1 (by committing a Data sample with Data.inlineQos.groupSequenceNumber >= GSN) to the Subscriber history cache or a Gap message with Gap.gapEndGSN.value >= GSN-1.
- OR announced it does not have the GSN-1 by sending a Heartbeat with Heartbeat.currentGSN.value >= GSN and GSN-1 \notin [ _Heartbeat.firstGSN.value_, _Heartbeat.lastGSN.value_ ].

The above rules should only take into consideration DataWriters that have not lost their liveliness, see 8.7.2.2.3. Implementations could use additional timeout-based rules to limit delays.

### 8.7.6 Coherent Sets

The DDS specification provides the functionality to define a set of sample updates as a coherent set. A DataReader is only notified of the arrival of new updates once all updates in the coherent set have been received.

A “Publisher coherent set” is defined as the set of all CacheChanges performed by all DataWriters in the Publisher delimited by the operations begin_coherent_changes() and end_coherent_changes().

Resulting from each “Publisher coherent set” there may be one or more “Subscriber coherent sets” defined for each Subscriber in the system. What constitutes a “Subscriber coherent set” depends on the PRESENTATION access_scope of the Subscriber:

- If the Subscriber has PRESENTATION coherent_access=FALSE then there are no Subscriber coherent sets. Alternatively, this could be interpreted as if each individual CacheChange was an independent Subscriber coherent set.
- If the Subscriber has PRESENTATION access_scope=INSTANCE or TOPIC then there is a separate “Subscriber” coherent set for each DataWriter containing the subset of samples that are written by each of the DataWriters in the Publisher.
- If the Subscriber has PRESENTATION access_scope=GROUP then the Subscriber coherent set matches the Publisher coherent set.

A “Subscriber-relevant coherent set” is the subset of changes in the “Subscriber coherent set” that the Subscriber must receive in order to consider the coherent set complete. Incomplete coherent sets shall not be added to the history of the RTPS DataReaders and the corresponding CacheChanges shall be discarded by the Subscriber.

The “Subscriber-relevant coherent set” is defined as the subset of the “Subscriber coherent change” obtained after removing the following CacheChanges:

- Changes that belong to DataWriters that are not matched with corresponding DataReaders in the Subscriber.
- Changes that are filtered by content or time.

Note that samples replaced due to history depth are considered part of the “Subscriber-relevant coherent set” if any is not received the coherent set is not complete. Likewise, for samples lost due to the use of best-effort protocol or other reasons.
In order to support coherent sets, RTPS uses the in-line QoS parameter extension mechanism to include additional information in-line with each Data Submessage. The additional information denotes membership to a particular coherent set. The remainder of the discussion only refers to Data Submessages, but the same approach is followed for DataFrag Submessages.

For access scope TOPIC, all Data Submessages belonging to the same coherent set have strict monotonically increasing sequence numbers (as they originated from the same Writer). Therefore, a coherent set is uniquely identified by the sequence number of the first sample update belonging to the coherent set. All sample updates belonging to the same coherent set contain an in-line QoS parameter with this same sequence number. This approach also allows the Reader to easily determine when the coherent set started.

The end of a Writer’s coherent set is defined by the arrival of one of the following:

- A Data Submessage from this Writer that belongs to a new coherent set.
- A Data Submessage from this Writer that does not contain a coherent set in-line QoS parameter or alternatively, contains a coherent set in-line QoS parameter with value SEQUENCENUMBER_UNKNOWN. Both approaches are equivalent.

Note that a Data Submessage need not necessarily contain serializedPayload. This makes it possible to notify the Reader about the end of a coherent set before the next data is written by the Writer.

For access scope GROUP, all Data submessages and the first DataFrag submessage belonging to the same coherent set have strictly monotonically increasing group sequence numbers (as they originated from the same Publisher). Therefore, a group coherent set is uniquely identified by the group sequence number of the first sample belonging to the coherent set. All Data submessages and the first DataFrag submessage belonging to the same group coherent set shall have three in-line QoS parameters:

- The PID_GROUP_SEQ_NUM shall contain the group sequence number.
- The PID_COHERENT_SET shall contain the sequence number of the first sample update belonging to the coherent set from the Writer.
- The PID_GROUP_COHERENT_SET shall contain the group sequence number of the first sample update belonging to the coherent set across all Writers within the Publisher.

A group’s coherent set is marked as being finished by sending an End Coherent Set (ECS) Data submessage from all Writers within the Publisher. The ECS Data Submessage shall have the following properties:

- It does not contain a serializedPayload
- Its group sequence number is equal to one greater than the group sequence number of the final sample in the group coherent set.
- It is not filtered by time, content, history, lifespan, etc. It can only be removed from the RTPS Writer cache when all data samples belonging to the coherent set are removed.
- It does not count towards resource limits.
- It has the InlineQos parameters PID_GROUP_SEQ_NUM, PID_GROUP_COHERENT_SET, PID_WRITER_GROUP_INFO.
- If required, it may also contain PID_SECURE_WRITER_GROUP_INFO. See section 9.6.4.5 for details.

The ECS Data Submessage is sent with in-line QoS parameters:

- PID_GROUP_SEQ_NUM: The group sequence number one greater than the group sequence number of the last sample in the coherent set.
- PID_GROUP_COHERENT_SET: The group sequence number of the coherent set that it marks the end of.
- PID_GROUP_WRITER_INFO: The writer group information encoding which writers were contained in the Publisher during the time that the coherent set was written. Note that Writers are not allowed to be added or removed from a Publisher from the time that a coherent set begins until after it ends.
A **DataReader** that receives samples in a group coherent set first waits for the complete coherent set from each remote **DataWriter** separately. Once a coherent set from a **DataWriter** is complete, the **DataReader** commits the entire set to the **HistoryCache** of the **Subscriber**. The **Subscriber** orders these individual coherent sets from each **DataReader** according to the same rules that are applied for ordered access with scope set to **GROUP**. The group coherent set becomes ready to be committed to the DDS **DataReader** once an ECS sample is committed to the **Subscriber** and the ECS sample meets the criteria for being committed to the DDS **DataReader**.

Once the group coherent set becomes ready to be committed the **Subscriber** shall determine if the subscriber-relevant coherent set is complete and if so, make it available to the application.

### 8.7.7 Directed Write

Direct peer-to-peer communications where a Writer explicitly identifies a Subset of its matched Readers as the intended destination for a particular sample is useful in some application scenarios.

RTPS supports directed writes by using the in-line QoS parameter extension mechanism. The serialized information denotes the GUIDs of the targeted reader(s).

When a writer sends a directed sample, only recipients with a matching GUID accept the sample; all other recipients acknowledge but absorb the sample, as if it were a GAP message.

### 8.7.8 Property Lists

Property lists are lists of user-definable properties applied to a DDS Entity. An entry in the list is a generic name-value pair. A user defines a pair to be a property for a DDS Participant, DataWriter, or DataReader. This extensible list enables non-DDS-specified properties to be applied.

The RTPS protocol supports Property Lists as in-line parameters. Properties can then be propagated during Discovery or as in-line QoS.

### 8.7.9 Original Writer Info

A service supporting the TransientLocal, Transient, or Persistent level of DDS Durability QoS needs to send the data that has been received and stored on behalf of the persistent writer.

This service that forwards messages needs to indicate that the forwarded message belongs to the message-stream of another writer, such that if the reader receives the same messages from another source (for example, another forwarding service or the original writer), it can treat them as duplicates.

The RTPS protocol supports this forwarding of messages by including information of the original writer.

When a RTPS Reader receives this information, it will treat it as a normal CacheChange, but once the CacheChange is ready to be committed to the DDS DataReader, it will not commit it. Instead, it will hand if off to the HistoryCache of the RTPS Reader that is communicating with the RTPS Writer indicated in the ORIGINAL_WRITER_INFO in-line QoS and treat is as having the sequence number which appears there.

#### Table 8.89 - Original writer info

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>originalWriterGUID</td>
<td>GUID_t</td>
<td>The GUID of the RTPS Writer that first generated the message.</td>
</tr>
<tr>
<td>originalWriterSN</td>
<td>SequenceNumber_t</td>
<td>The Sequence Number of the CacheChange as sent from the original writer.</td>
</tr>
</tbody>
</table>
8.7.10 Key Hash

The Key Hash provides a hint for the key that uniquely identifies the data-object that is being changed within the set of objects that have been registered by the DDS DataWriter.

Nominally the key is part of the serialized data of a data submessage. Using the key hash benefits implementations by providing a faster alternative than deserializing the full key from the received data-object.

When the key hash is not received by a DataReader, it should be computed from the data itself. If there is no data in the submessage, then a default zero-valued key hash should be used by the DataReader.

A Key Hash, if present, shall be computed as described in 9.6.4.3.
9 Platform Specific Model (PSM): UDP/IP

9.1 Introduction

This clause defines the Platform Specific Model (PSM) that maps the Protocol PIM to UDP/IP. The goal for this PSM is to provide a mapping with minimal overhead directly on top of UDP/IP.

The suitability of UDP/IP as a transport for DDS applications stems from several factors:

- Universal availability. Being a core part of the IP stack, UDP/IP is available on virtually all operating systems.
- Light-weight. UDP/IP is a very simple protocol that adds minimal services on top of IP. Its use enables the use of IP-based networks with the minimal possible overhead.
- Best-effort. UDP/IP provides a best-effort service that maps well to Quality-of-service needs of many real-time data streams. In the situations where it is needed, the RTPS protocol provides the mechanism to attain reliable delivery on top of the best-effort service provided by UDP.
- Connectionless. UDP/IP offers a connectionless service; this allows multiple RTPS endpoints to share a single operating system UDP resource (socket/port) while allowing for interleaving of messages effectively providing an out-of-band mechanism for each separate data-stream.
- Predictable behavior. Unlike TCP, UDP does not introduce timers that would cause operations to block for varying amounts of time. As such, it is simpler to model the impact of using UDP on a real-time application.
- Scalability and multicast support. UDP/IP natively supports multicast which allows efficient distribution of a single message to a large number of recipients.

9.2 Notational Conventions

9.2.1 Name Space

All the definitions in this document are part of the “RTPS” name-space. To facilitate reading and understanding, the name-space prefix has been left out of the definitions and classes in this document.

9.2.2 IDL Representation of Structures and CDR Wire Representation

The following sub clauses often define structures, such as:

```c
typedef octet OctetArray3[3];

struct EntityId_t {
    OctetArray3 entityKey;
    octet entityKind;
};
```

These definitions use the OMG IDL (Interface Definition Language). When these structures are sent on the wire, they are encoded using the corresponding CDR representation.

9.2.3 Representation of Bits and Bytes

This document often uses the following notation to represent an octet or byte:

```
+----------------------------------+
|7|6|5|4|3|2|1|0|
+----------------------------------+
```

In this notation, the leftmost bit (bit 7) is the most significant bit (“MSB”) and the rightmost bit (bit 0) is the least significant bit (“LSB”).

Streams of bytes are ordered per lines of 4 bytes each as follows:
The information on the kind of entity encoded in the last six bits of the entityKind field. These two bits are set to:

- ‘8’ for user-defined entities.
- ‘11’ for built-in entities.
- ‘01’ for vendor-specific entities.

The information on the kind of Entity is encoded in the last six bits of the entityKind field. Table 9.1 provides a complete list of the possible values of the entityKind supported in version 2.5 of the protocol. These are fixed for user-defined entities. The target values for the entityKind field for user-defined entities are ‘79’ (Protocol), ‘80’ (Reader), ‘81’ (Writer), ‘82’ (Writer Group), and ‘83’ (Reader Group). The other bits in the entityKind field can be used to encode information specific to the protocol, such as the version number, but these are not specified in the standard. The remaining bits are reserved for future use. The reserved constant ENTITYID_UNKNOWN defined by the PIM is mapped to:

```
#define ENTITYID_UNKNOWN {{0x00, 0x00, 0x00}, 0x00}
```

The reserved constant GUIDPREFIX_UNKNOWN defined by the PIM is mapped to:

```
#define GUIDPREFIX_UNKNOWN {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}
```

In this representation, the byte that comes first in the stream is on the left. The bit on the extreme left is the MSB of the first byte; the bit on the extreme right is the LSB of the 4th byte.

9.3 Mapping of the RTPS Types

9.3.1 The Globally Unique Identifier (GUID)

The GUID is an attribute present in all RTPS Entities that uniquely identifies them within the DDS domain (see 8.2.4.1). The PIM defines the GUID as composed of a GuidPrefix_t prefix capable of holding 12 bytes, and an EntityId_t entityId capable of holding 4 bytes. This sub clause defines how the PSM maps those structures.

9.3.1.1 Mapping of the GuidPrefix_t

The PSM maps the GuidPrefix_t to the following structure:

```
typedef octet GuidPrefix_t[12];
```

The reserved constant GUIDPREFIX_UNKNOWN defined by the PIM is mapped to:

```
#define GUIDPREFIX_UNKNOWN {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}
```

9.3.1.2 Mapping of the EntityId_t

Clause 8.2.4.3 states that the EntityId_t is the unique identification of the Endpoint within the Participant. The PSM maps the EntityId_t to the following structure:

```
typedef octet OctetArray3[3];
struct EntityId_t {
  OctetArray3 entityKey;
  octet entityKind;
};
```

The reserved constant ENTITYIDUNKNOWN defined by the PIM is mapped to:

```
#define ENTITYIDUNKNOWN {{0x00, 0x00, 0x00}, 0x00}
```

The entityKind field within EntityId_t encodes the kind of Entity (Participant, Reader, Writer, Reader Group, Writer Group) and whether the Entity is a built-in Entity (fully pre-defined by the Protocol, automatically instantiated), a user-defined Entity (defined by the Protocol, but instantiated by the user only as needed by the application) or a vendor-specific Entity (defined by a vendor-specific extension to the Protocol, can therefore be ignored by another vendor’s implementation).

When not pre-defined (see below), the entityKey field within EntityId_t can be chosen arbitrarily by the middleware implementation as long as the resulting EntityId_t is unique within the Participant.

The information on whether the object is a built-in entity, a vendor-specific entity, or a user-defined entity is encoded in the two most-significant bits of the entityKind. These two bits are set to:

- ‘8’ for user-defined entities.
- ‘11’ for built-in entities.
- ‘01’ for vendor-specific entities.

The information on the kind of Entity is encoded in the last six bits of the entityKind field. Table 9.1 provides a complete list of the possible values of the entityKind supported in version 2.5 of the protocol. These are fixed
in this major version (2) of the protocol. New entity Kinds may be added in higher minor versions of the protocol in order to extend the model with new kinds of Entities.

Table 9.1 - entityKind octet of an EntityId_t

<table>
<thead>
<tr>
<th>Kind of Entity</th>
<th>User-defined Entity</th>
<th>Built-in Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>unknown</td>
<td>0x00</td>
<td>0xc0</td>
</tr>
<tr>
<td>Participant</td>
<td>N/A</td>
<td>0xc1</td>
</tr>
<tr>
<td>Writer (with Key)</td>
<td>0x02</td>
<td>0xc2</td>
</tr>
<tr>
<td>Writer (no Key)</td>
<td>0x03</td>
<td>0xc3</td>
</tr>
<tr>
<td>Reader (no Key)</td>
<td>0x04</td>
<td>0xc4</td>
</tr>
<tr>
<td>Reader (with Key)</td>
<td>0x07</td>
<td>0xc7</td>
</tr>
<tr>
<td>Writer Group</td>
<td>0x08</td>
<td>0xc8</td>
</tr>
<tr>
<td>Reader Group</td>
<td>0x09</td>
<td>0xc9</td>
</tr>
</tbody>
</table>

### 9.3.1.3 Predefined EntityIds

As mentioned above, the entity IDs for built-in entities are fully predefined by the RTPS Protocol.

The PIM specifies that the EntityId_t of a Participant has the pre-defined value ENTITYID_PARTICIPANT (8.2.4.2). The corresponding PSM mapping of all pre-defined Entity IDs appears in Table 9.2 - EntityId_t values fully predefined by the RTPS Protocol. The meaning of these Entity IDs cannot change in this major version (2) of the protocol, but future minor versions may add additional reserved Entity IDs.

Table 9.2 - EntityId_t values fully predefined by the RTPS Protocol

<table>
<thead>
<tr>
<th>Entity</th>
<th>Corresponding value for entityId_t (NAME = value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>participant</td>
<td>ENTITYID_PARTICIPANT = {00,00,01},0xc1</td>
</tr>
<tr>
<td>SEDPbuiltinTopicWriter</td>
<td>ENTITYID_SEDP_BUILTIN_TOPICS_ANNOUNCER = {00,00,02},0xc2</td>
</tr>
<tr>
<td>SEDPbuiltinTopicReader</td>
<td>ENTITYID_SEDP_BUILTIN_TOPICS_DETECTOR = {00,00,02},0xc7</td>
</tr>
<tr>
<td>SEDPbuiltinPublicationsWriter</td>
<td>ENTITYID_SEDP_BUILTIN_PUBLICATIONS_ANNOUNCER = {00,00,03},0xc2</td>
</tr>
<tr>
<td>SEDPbuiltinPublicationsReader</td>
<td>ENTITYID_SEDP_BUILTIN_PUBLICATIONS_DETECTOR = {00,00,03},0xc7</td>
</tr>
<tr>
<td>SEDPbuiltinSubscriptionsWriter</td>
<td>ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_ANNOUNCER = {00,00,04},0xc2</td>
</tr>
<tr>
<td>SEDPbuiltinSubscriptionsReader</td>
<td>ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_DETECTOR = {00,00,04},0xc7</td>
</tr>
<tr>
<td>SPDPbuiltinParticipantWriter</td>
<td>ENTITYID_SPDP_BUILTIN_PARTICIPANT_ANNOUNCER = {00,01,00},0xc2</td>
</tr>
<tr>
<td>SPDPbuiltinParticipantReader</td>
<td>ENTITYID_SPDP_BUILTIN_PARTICIPANT_DETECTOR = {00,01,00},0xc7</td>
</tr>
<tr>
<td>BuiltinParticipantMessageWriter</td>
<td>ENTITYID_P2P_BUILTIN_PARTICIPANT_MESSAGE_WRITER = {00,02,00},0xc2</td>
</tr>
<tr>
<td>BuiltinParticipantMessageReader</td>
<td>ENTITYID_P2P_BUILTIN_PARTICIPANT_MESSAGE_READER = {00,02,00},0xc7</td>
</tr>
</tbody>
</table>
9.3.1.3.1 EntityIds Reserved by other Specifications

Other specifications may reserve EntityIds. Table 9.3 lists the EntityIds reserved for use by other specifications and future revisions thereof.

Table 9.3 - EntityIds Reserved by other Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Reserved EntityId</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS-Security 1.1</td>
<td>EntityIds that have both an entityKey in the range {ff, 00, 00} – {ff, ff, ff} and an entityKind in the range 0xc0-0xff (inclusive).</td>
</tr>
<tr>
<td>DDS-Security 1.1</td>
<td>{{00, 02, 01}, c3} and {{00, 02, 01}, c4}</td>
</tr>
<tr>
<td>DDS-XTypes 1.2</td>
<td>{{00, 03, 00}, c3}, {{00, 03, 00}, c4}, {{00, 03, 01}, c3}, {{00, 03, 01}, c4}</td>
</tr>
<tr>
<td>DDS-XTypes 1.3</td>
<td>{{00, 03, 00}, c3}, {{00, 03, 00}, c4}, {{00, 03, 01}, c3}, {{00, 03, 01}, c4}</td>
</tr>
</tbody>
</table>

9.3.1.4 Deprecated EntityIds in version 2.2 of the Protocol

The Discovery Protocol used in version 2.2 of the protocol deprecates the EntityIds shown in Table 9.4 - Deprecated EntityIds in version 2.2 of the protocol. These EntityIds should not be used by future versions of the protocol unless they are used with the same meaning as in versions prior to 2.2. Implementations that wish to discover earlier versions should utilize these EntityIds.

Table 9.4 - Deprecated EntityIds in version 2.2 of the protocol

<table>
<thead>
<tr>
<th>Entity</th>
<th>Corresponding entityId</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>0x05</td>
</tr>
<tr>
<td>Server</td>
<td>0x06</td>
</tr>
<tr>
<td>writerApplications</td>
<td>{{00,00,01},c2}</td>
</tr>
<tr>
<td>readerApplications</td>
<td>{{00,00,01},c7}</td>
</tr>
<tr>
<td>writerClients</td>
<td>{{00,00,05},c2}</td>
</tr>
<tr>
<td>readerClients</td>
<td>{{00,00,05},c7}</td>
</tr>
<tr>
<td>writerServices</td>
<td>{{00,00,06},c2}</td>
</tr>
<tr>
<td>readerServices</td>
<td>{{00,00,06},c7}</td>
</tr>
<tr>
<td>writerManagers</td>
<td>{{00,00,07},c2}</td>
</tr>
<tr>
<td>readerManagers</td>
<td>{{00,00,07},c7}</td>
</tr>
<tr>
<td>writerApplicationsSelf</td>
<td>{{00,00,08},c2}</td>
</tr>
</tbody>
</table>

9.3.1.5 Mapping of the GUID_t

The PSM maps the GUID_t to the following structure:

```c
struct GUID_t {
    GuidPrefix_t guidPrefix;
    EntityId_t entityId;
};
```

Sub clause 8.2.4 states that all RTPS Entities with a DomainParticipant share the same guidPrefix. Furthermore 8.2.4.2 states that implementors have freedom to choose the guidPrefix as long as each DomainParticipant within a DDS Domain has a unique guidPrefix. The PIM restricts this freedom.

To comply with this specification, implementations of the RTPS protocol shall set the first two bytes of the guidPrefix to match their assigned vendorId (see 8.3.3.1.3). This ensures that the guidPrefix remains unique within a DDS Domain even if multiple implementations of the protocol are used. In other words,
implementations of the RTPS protocol are free to use any technique they deem appropriate to generate unique values for the `guidPrefix` as long as they meet the following constraint:

\[
guidPrefix[0] = \text{vendorId}[0], \quad guidPrefix[1] = \text{vendorId}[1]
\]

Future versions of the RTPS 2.x protocol shall also follow this rule for generating the `guidPrefix`.

The value of these first two bytes is set as specified above with the sole purpose of enabling the generation of unique `guidPrefix` across implementations. This value should not be relied upon for other purposes. This ensures the change does not break interoperability with previous versions of the protocol.

Use of the reserved `vendorId` is further described in 9.4.4.

The reserved constant `GUID_UNKNOWN` defined by the PIM is mapped to:

```c
#define GUID_UNKNOWN { GUIDPREFIX_UNKNOWN, ENTITYID_UNKNOWN }
```

### 9.3.2 Mapping of the Types that Appear Within Submessages or Built-in Topic Data

#### 9.3.2.1 IDL Definitions

The following IDL specifies the PSM mapping of the types that are introduced by the PIM that appear within messages sent by the protocol. There is no need to map the types that are used exclusively by the virtual machine, but do not appear in the messages. The subsections following the IDL provide additional information for the mapped types which require further clarification beyond the IDL type.

```c
typedef unsigned long DomainId_t;

// TIME_ZERO: seconds = 0, fraction = 0
// TIME_INVALID: seconds = 0xffffffff, fraction = 0xffffffff
// TIME_INFINITE: seconds = 0xffffffff, fraction = 0xfffffffe
struct Time_t {
    unsigned long seconds; // time in seconds
    unsigned long fraction; // time in sec/2^32
};

// DURATION_ZERO: seconds = 0, fraction = 0
// DURATION_INFINITE: seconds = 0x7fffffff, fraction = 0xffffffff
struct Duration_t {
    long seconds; // time in seconds
    unsigned long fraction; // time in sec/2^32
};

// VENDORID_UNKNOWN: VendorId_t[0] = 0, VendorId_t[1] = 0
typedef octet VendorId_t[2];

// SEQUENCENUMBER_UNKNOWN: high = -1, low = 0
// Using this structure, the 64-bit sequence number is:
// seq_num = high * 2^32 + low
struct SequenceNumber_t {
    long high;
    unsigned long low;
};

struct ChangeCount_t {
    long high;
    unsigned long low;
};

typedef unsigned long FragmentNumber_t;

const long LOCATOR_KIND_INVALID = -1;
const long LOCATOR_KIND_RESERVED = 0;
```
const long LOCATOR_KIND_UDPv4 = 1;
const long LOCATOR_KIND_UDPv6 = 2;
const unsigned long LOCATOR_PORT_INVALID = 0;

// LOCATOR_ADDRESS_INVALID: {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}
// LOCATOR_INVALID: kind = LOCATOR_KIND_INVALID
//                  port = LOCATOR_PORT_INVALID
//                  address = LOCATOR_ADDRESS_INVALID
struct Locator_t {
    long kind;
    unsigned long port;
    octet address[16];
};

// The values of the following constants as defined in the DDS Specification
// should be mapped to the below values before being sent on the wire.
const long BEST_EFFORT = 1;
const long RELIABLE = 2;
typedef long ReliabilityKind_t;

typedef long Count_t;

// The implementations following this version of the document
// implement protocol version 2.4
struct ProtocolVersion_t {
    octet major;
    octet minor;
};
typedef octet KeyHash_t[16];
typedef octet StatusInfo_t[4];
typedef short ParameterId_t;

struct ContentFilterProperty_t {
    string<256> contentFilteredTopicName;
    string<256> relatedTopicName;
    string<256> filterClassName;
    string filterExpression;
    sequence<string> expressionParameters;
};
typedef sequence<long> FilterResult_t;
typedef long FilterSignature_t[4];
typedef sequence<FilterSignature_t> FilterSignatureSequence;
struct ContentFilterInfo_t {
    FilterResult_t filterResult;
    FilterSignatureSequence filterSignatures;
};

struct Property_t {
    string name;
    string value;
};
typedef string EntityName_t;

struct OriginalWriterInfo_t {
    GUID_t originalWriterGUID;
    SequenceNumber_t originalWriterSN;
    ParameterList originalWriterQos;
};
typedef octet GroupDigest_t[4];

/* The following bitmask identifies protocol-specific builtin endpoints. Vendor-specific builtin endpoints may be identified by a new vendor-specific ParameterId. Refer to section 9.6.2.2.1 ParameterId space for the range of ParameterIds that are available for vendor-specific extensions. */

bitmask BuiltinEndpointSet_t {
    @position(0) DISC_BUILTIN_ENDPOINT_PARTICIPANT_ANNOUNCER,
    @position(1) DISC_BUILTIN_ENDPOINT_PARTICIPANT_DETECTOR,
    @position(2) DISC_BUILTIN_ENDPOINT_PUBLICATIONS_ANNOUNCER,
    @position(3) DISC_BUILTIN_ENDPOINT_PUBLICATIONS_DETECTOR,
    @position(4) DISC_BUILTIN_ENDPOINT_SUBSCRIPTIONS_ANNOUNCER,
    @position(5) DISC_BUILTIN_ENDPOINT_SUBSCRIPTIONS_DETECTOR,

    /* The following have been deprecated in version 2.4 of the specification. These bits should not be used by versions of the protocol equal to or newer than the deprecated version unless they are used with the same meaning as in versions prior to the deprecated version. */
    @position(6) DISC_BUILTIN_ENDPOINT_PARTICIPANT_PROXY_ANNOUNCER,
    @position(7) DISC_BUILTIN_ENDPOINT_PARTICIPANT_PROXY_DETECTOR,
    @position(8) DISC_BUILTIN_ENDPOINT_PARTICIPANT_STATE_ANNOUNCER,
    @position(9) DISC_BUILTIN_ENDPOINT_PARTICIPANT_STATE_DETECTOR,

    @position(10) BUILTIN_ENDPOINT_PARTICIPANT_MESSAGE_DATA_WRITER,
    @position(11) BUILTIN_ENDPOINT_PARTICIPANT_MESSAGE_DATA_READER,

    /* Bits 12-15 have been reserved by the DDS-Xtypes 1.2 Specification and future revisions thereof. Bits 16-27 have been reserved by the DDS-Security 1.1 Specification and future revisions thereof. */
    @position(12) DISC_BUILTIN_ENDPOINT_TOPICS_ANNOUNCER,
    @position(13) DISC_BUILTIN_ENDPOINT_TOPICS_DETECTOR
};

bitmask BuiltinEndpointQos_t {
    @position(0) BEST_EFFORT_PARTICIPANT_MESSAGE_DATA_READER
};

// PROTOCOL_RTPS:
/* ProtocolId_t[0] = 'R'
 * ProtocolId_t[1] = 'T'
 * ProtocolId_t[2] = 'P'
 * ProtocolId_t[3] = 'S'
 */
typedef octet ProtocolId_t[4];

// RTPS HeaderExtension
typedef unsigned long MessageLength_t;
const MessageLength_t MESSAGE_LENGTH_INVALID = 0;

typedef octet UExtension4_t[4];
typedef octet WExtension8_t[8];
typedef octet Checksum32_t[4];
typedef octet Checksum64_t[8];
typedef octet Checksum128_t[16];
9.3.2.2 Time_t

The representation of the time is the one defined by the IETF Network Time Protocol (NTP) Standard (IETF RFC 1305). In this representation, time is expressed in seconds and fractions of seconds using the formula:
\[
time = \text{seconds} + (\text{fraction} / 2^{32})
\]

9.3.2.3 Duration_t

The representation of the time is the one defined by the IETF Network Time Protocol (NTP) Standard (IETF RFC 1305). In this representation, time is expressed in seconds and fractions of seconds using the formula:
\[
time = \text{seconds} + (\text{fraction} / 2^{32})
\]

Versions of the RTPS specification previous to version 2.4 did not specify the representation of Duration_t, therefore implementations should take into account the vendor and protocol version when interpreting these fields.

9.3.2.4 Locator_t

If the Locator_t kind is LOCATOR_KIND_UPDv4, the address contains an IPv4 address. In this case, the leading 12 octets of the address must be zero. The last 4 octets are used to store the IPv4 address. The mapping between the dot-notation “a.b.c.d” of an IPv4 address and its representation in the address field of a Locator_t is:
\[
\text{address} = (0,0,0,0,0,0,0,0,0,0,0,\text{a,b,c,d})
\]

If the Locator_t kind is LOCATOR_KIND_UPDv6, the address contains an IPv6 address. IPv6 addresses typically use a shorthand hexadecimal notation that maps one-to-one to the 16 octets in the address field. For example, the representation of the IPv6 address “FF00:4501:0:0:0:0:0:32” is:
\[
\text{address} = (0xff,0,0x45,0x01,0,0,0,0,0,0,0,0,0,0,0,0x32)
\]

The range of Locator_t kinds has been divided into the following ranges:
- 0x00000003 - 0x01ffffff (inclusive) are reserved for vendor-specific Locator_t kinds and will not be used by any future versions of the RTPS protocol.
- 0x02000000 - 0x02ffffff (inclusive) are reserved for future use by the RTPS specification
- 0x03000000 and greater are reserved for Locator_t kinds that identify a transport developed by a third-party (i.e., are neither vendor nor protocol-specific) and will not be used by any future versions of the RTPS protocol.

9.3.2.5 GroupDigest_t

This type is used to represent a group of Entities belonging to the same Participant. The representation uses the IDL structure EntityIdSet_t defined below:
\[
\text{typedef octet OctetArray3[3];}
\text{struct {}
    OctetArray3 entityKey;
    octet entityKind;
};
\text{struct EntityIdSet_t {}
    sequence<EntityId_t> entityIds;
};
\]

In the construction of the entityIds sequence, the values are sorted by increasing values of the EntityId_t. To perform the ordering the EntityId_t, which is 4 octets, is re-interpreted as if it was the little-endian serialized representation of a 32-bit signed integer (the IDL4 int32 primitive type).

The GroupDigest_t is computed from an EntityIdSet_t by first computing a 128 bit MD5 Digest (IETF RFC 1321) applied to the CDR Big-Endian serialization of the structure EntityIdSet_t. The GroupDigest_t is the leading 4 octets of the MD5 Digest.

The empty group is represented by a zero value of the GroupDigest_t. It is not computed as the hash of the serialized empty sequence.
9.3.2.6 Checksum32_t, Checksum64_t, Checksum128_t
These types are used to represent checksums of various lengths: Checksum32_t represents a 32-bit checksum. Checksum64_t, and Checksum128_t represent a 64-bit, and 128-bit checksum, respectively.

typedef octet Checksum32_t[4];
typedef octet Checksum64_t[8];
typedef octet Checksum128_t[16];

9.3.2.7 MessageLength_t
This type is used to represent the length of an RTPS message. The representation uses a 32-bit unsigned integer.

typedef unsigned long MessageLength_t;

9.3.2.8 UExtension4_t
This type is used to represent an undefined 4-byte value.

typedef octet UExtension4_t;

9.3.2.9 WExtension8_t
This type is used to represent an undefined 8-byte value.

typedef octet WExtension8_t;

9.3.2.10 SequenceNumber_t
This type is used to represent a 64-bit sequence number.

The sequence number is represented using a structure that contains two 32-bit integers: high and low.

struct SequenceNumber_t {
  long high;
  unsigned long low;
};

The 64-bit sequence number is obtained using the formula:

 sequence_number = low + high * 2^(32)

9.3.2.11 ChangeCount_t
This type is used to represent a 64-bit count.

The change count is represented using a structure that contains two 32-bit integers: high and low.

struct ChangeCount_t {
  long high;
  unsigned long low;
};

The 64-bit count is obtained using the formula:

 change_count = low + high * 2^(32)

9.3.2.12 BuiltinEndpointSet_t
This type is used to represent a list of builtin endpoints.

The set of endpoints is represented using a bitmap. Each bit in the bitmap represents a specific builtin endpoint:

 bitmask BuiltinEndpointSet_t {
   @position(0) DISC_BUILTIN_ENDPOINT_PARTICIPANT_ANNOUNCER,
defined as defined in the underlying transport with which RTPS is deployed

Each RTPS Message is built from a set of predefined atomic building blocks called “submessage elements,” as defined in 8.3.5. This sub clause describes the PSM mapping for each of the SubmessageElements defined by the PIM.

Other DDS specifications may also define builtin endpoints and may communicate their presence setting bits within the BuiltinEndpointsSet_t:

- Positions 12-15 are reserved by DDS-XTypes 1.2 and its revisions thereof, see DDS-XTypes 1.2 clause 7.6.2.3.4 and DDS-XTypes 1.3 clause 7.6.3.3.4.
- Positions 16-27 are reserved by DDS-Security 1.1 and revisions thereof, see DDS-Security 1.1 clause 7.4.1.4.

9.4 Mapping of the RTPS Messages

9.4.1 Overall Structure

Sub clause 8.3.3 in the PIM defined the overall structure of a Message as composed of a leading Header followed by a variable number of Submessages.

The PSM aligns each Submessage on a 32-bit boundary with respect to the start of the Message.

A Message has a well-known length. This length is not sent explicitly by the RTPS protocol but is part of the underlying transport with which Messages are sent. In the case of UDP/IP, the length of the Message is the length of the UDP payload.

9.4.2 Mapping of the PIM SubmessageElements

Each RTPS Submessage is built from a set of predefined atomic building blocks called “submessage elements,” as defined in 8.3.5. This sub clause describes the PSM mapping for each of the SubmessageElements defined by the PIM.
9.4.2.1 EntityId

The PSM mapping for the **EntityId** SubmessageElement defined in 8.3.5.1 is given by the following IDL definition:

```idl
typedef EntityId_t EntityId;
```

Following the CDR encoding, the wire representation of the **EntityId** SubmessageElement is:

```pseudocode
EntityId:
0...2...........8...............16............24............32
+---------------------------------+-------------------------+-------------------------+
|                                   | octet value[4]          |
|                                   | +-------------------------+
|                                   +-------------------------+
```

9.4.2.2 GuidPrefix

The PSM mapping for the **GuidPrefix** SubmessageElement defined in 8.3.5.1 is given by the following IDL definition:

```idl
typedef GuidPrefix_t GuidPrefix;
```

Following the CDR encoding, the wire representation of the **GuidPrefix** SubmessageElement is:

```pseudocode
GuidPrefix:
0...2...........8...............16............24............32
+---------------------------------+-------------------------+-------------------------+
|                                   | octet value[12]         |
|                                   | +-------------------------+
|                                   +-------------------------+
```

9.4.2.3 VendorId

The PSM mapping for the **VendorId** SubmessageElement defined in 8.3.5.2 is given by the following IDL definition:

```idl
typedef VendorId_t VendorId;
```

Following the CDR encoding, the wire representation of the **VendorId** SubmessageElement is:

```pseudocode
VendorId:
0...2...........8...............16
+---------------------------------+-------------------------+
|                                   | octet vendorId[2]       |
|                                   +-------------------------+
```

9.4.2.4 ProtocolVersion

The PSM mapping for the **ProtocolVersion** SubmessageElement defined in 8.3.5.3 is given by the following IDL definition:

```idl
typedef ProtocolVersion_t ProtocolVersion;
```

Following the CDR encoding, the wire representation of the **ProtocolVersion** SubmessageElement is:
9.4.2.5 SequenceNumber

The PSM mapping for the SequenceNumber SubmessageElement defined in 8.3.5.4 is given by the following IDL definition:

```idl
type SequenceNumber_t
SequenceNumber;
```

Following the CDR encoding, the wire representation of the SequenceNumber SubmessageElement is:

```plaintext
| long high | | unsigned long low |
```

9.4.2.6 SequenceNumberSet

The PSM maps the SequenceNumberSet SubmessageElement defined in 8.3.5.5 to the following structure:

```c
typedef sequence<long, 8> LongSeq8;

struct SequenceNumberSet {
    SequenceNumber_t bitmapBase;
    LongSeq8 bitmap;
};
```

The above structure offers a compact representation encoding a set of up to 256 sequence numbers. The representation of the SequenceNumberSet includes the first sequence number in the set (bitmapBase) and a bitmap of up to 256 bits. The number of bits in the bitmap is denoted by numBits. The value of each bit in the bitmap indicates whether the SequenceNumber obtained by adding the offset of the bit to the bitmapBase is included (bit=1) or excluded (bit=0) from the SequenceNumberSet.

More precisely a SequenceNumber 'seqNum' belongs to the SequenceNumberSet 'seqNumSet,' 'if and only if' the following two conditions apply:

```plaintext
seqNumSet.bitmapBase <= seqNum < seqNumSet.bitmapBase + seqNumSet.numBits (bitmap[deltaN/32] & (1 << (31 - deltaN%32))) == (1 << (31 - deltaN%32))
```

where

```plaintext
deltaN = seqNum - seqNumSet.bitmapBase
```

A valid SequenceNumberSet must satisfy the following conditions:

• bitmapBase >= 1
• 0 <= numBits <= 256
• there are M=(numBits+31)/32 longs containing the pertinent bits

This document uses the following notation for a specific bitmap:

```plaintext
bitmapBase/numBits:bitmap
```

In the bitmap, the bit corresponding to sequence number bitmapBase is on the left. The ending "0" bits can be represented as one "0."
For example, in bitmap “1234/12:00110”, bitmapBase=1234 and numBits=12. The bits apply as follows to the sequence numbers:

Table 9.5 - Example of bitmap: meaning of “1234/12:00110”

<table>
<thead>
<tr>
<th>SequenceNumber</th>
<th>Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>0</td>
</tr>
<tr>
<td>1235</td>
<td>0</td>
</tr>
<tr>
<td>1236</td>
<td>1</td>
</tr>
<tr>
<td>1237</td>
<td>1</td>
</tr>
<tr>
<td>1238-1245</td>
<td>0</td>
</tr>
</tbody>
</table>

The wire representation of the **SequenceNumberSet** SubmessageElement is:

```
SequenceNumberSet:
0...2...........8.................16.................24.................32
| +-----------------------------+-----------------------------+
| | SequenceNumber bitmapBase   |
| +-----------------------------+-----------------------------+
| | unsigned long numBits       |
| +-----------------------------+-----------------------------+
| | long bitmap[0]              |
| +-----------------------------+-----------------------------+
| | long bitmap[1]              |
| +-----------------------------+-----------------------------+
| | ...                         |
| +-----------------------------+-----------------------------+
| | long bitmap[M-1] M = (numBits+31)/32 |
```

The **numBits** field encodes both the number of significant bits and the number of bitmap elements. Due to this optimization, this SubmessageElement does not follow CDR encoding.

### 9.4.2.7 FragmentNumber

The PSM mapping for the **FragmentNumber** SubmessageElement defined in 8.3.5.6 is given by the following IDL definition:

```
typedef FragmentNumber_t FragmentNumber;
```

Following the CDR encoding, the wire representation of the **FragmentNumber** SubmessageElement is:

```
FragmentNumber:
0...2...........8.................16.................24.................32
| +-----------------------------+-----------------------------+
| | unsigned long value         |
```

### 9.4.2.8 FragmentNumberSet

The PSM maps the **FragmentNumberSet** SubmessageElement defined in 8.3.5.7 to the following structure:
typedef sequence<long, 8> LongSeq8; struct FragmentNumberSet {
    FragmentNumber_t bitmapBase; LongSeq8 bitmap;
};

The above structure offers a compact representation encoding a set of up to 256 fragment numbers. The representation of the FragmentNumberSet includes the first fragment number in the set (bitmapBase) and a bitmap of up to 256 bits. The interpretation matches that of a SequenceNumberSet.

The wire representation of the FragmentNumberSet SubmessageElement is:

<table>
<thead>
<tr>
<th>fragmentNumber bitmapBase</th>
<th>numBits</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitmap[0]</td>
<td></td>
</tr>
<tr>
<td>bitmap[1]</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>bitmap[M-1]</td>
<td>M = (numBits+31)/32</td>
</tr>
</tbody>
</table>

The numBits field encodes both the number of significant bits and the number of bitmap elements. Due to this optimization, this SubmessageElement does not follow CDR encoding.

9.4.2.9 Timestamp

The PSM mapping for the Timestamp SubmessageElement defined in 8.3.5.8 is given by the following IDL definition:

typedef Time_t Timestamp;

Following the CDR encoding, the wire representation of the Timestamp SubmessageElement is:

<table>
<thead>
<tr>
<th>long seconds</th>
<th>unsigned long fraction</th>
</tr>
</thead>
</table>

9.4.2.10 LocatorList

The PSM mapping for the LocatorList SubmessageElement defined in 8.3.5.16 is given by the following IDL definition:

typedef sequence<Locator_t, 8> LocatorList;

Following the CDR encoding, the wire representation of the LocatorList SubmessageElement is:
LocatorList:
0...2...........8..............16..............24. ............31
| ++........................++          |
| unsigned long numLocators |
| +-------------------------------+ |
| Locator_t locator_1 | ~ |
| ~ ..... ~ |
| Locator_t locator_numLocators |
| +-------------------------------+ |

Where each Locator_t has the following wire representation:
| ++........................++          |
| long kind |
| +-------------------------------+ |
| unsigned long port |
| +-------------------------------+ |
| octet address[16] |
| +-------------------------------+ |

9.4.2.11 ParameterList

A ParameterList contains a list of Parameters, each identified by a parameterId, optionally terminated with a sentinel.

9.4.2.11.1 Serialized Wire Representation

Each Parameter within the ParameterList starts aligned on a 4-byte boundary with respect to the start of the ParameterList.

The IDL representation for each Parameter is:

```c
typedef short ParameterId_t;
struct Parameter {
    ParameterId_t parameterId;
    short length;
    octet value[length]; // PseudoIDL: array of non-const length
};
```

The parameterId identifies the type of parameter.

The length encodes the number of octets following the length to reach the ID of the next parameter (or the ID of the sentinel). Because every parameterId starts on a 4-byte boundary, the length is always a multiple of four.

The value contains the CDR representation of the Parameter type that corresponds to the specified parameterId.

For alignment purposes, the CDR stream is logically reset for each parameter value (i.e., no initial padding is required) after the parameterId and length are serialized.

The ParameterList may contain multiple Parameters with the same value for the parameterId. This is used to provide a collection of values for that kind of Parameter.

The use of ParameterList representation makes it possible to extend the protocol and introduce new parameters and still be able to preserve interoperability with earlier versions of the protocol.

The wire representation for the ParameterList is:
There are two predefined values of the `parameterId`:

```c
#define PID_PAD (0)
#define PID_SENTINEL (1)
```

The PID_SENTINEL is used to terminate the parameter list and its length is ignored. The PID_PAD is used to enforce alignment of the parameter that follows and its length can be anything (as long as it is a multiple of 4).

The presence of the PID_SENTINEL is required in situations where it is not possible to determine the end of the ParameterList by some other mechanism.

- The presence of the PID_SENTINEL is not required in the ParameterList that appears in the HeaderExtension. See 9.4.5.2.
- The presence of the PID_SENTINEL is required in all other cases.

The complete set of possible values for the `parameterId` in version 2.5 of the protocol appears in 9.6.4.

### 9.4.2.11.2 ParameterId space

As described in 9.4.2.11.1, the ParameterId space is 16 bits wide. In order to accommodate vendor specific options and future extensions to the protocol, the ParameterId space is partitioned into multiple subspaces. The ParameterId subspaces are listed in Table 9.6.
Table 9.6 - ParameterId subspaces

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParameterId &amp; 8000 (Reserved or Vendor Specific)</td>
<td>0</td>
<td>Reserved ParameterId.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Vendor-specific ParameterId. Will not be recognized by other vendors’ implementations.</td>
</tr>
<tr>
<td>ParameterId &amp; 4000 (Ignore or Must Understand)</td>
<td>0</td>
<td>If the ParameterId is not recognized, skip and ignore the parameter.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>If the ParameterId is not recognized, treat it as an error. If the ParameterId appears in the HeaderExtension, ignore the entire RTPS message. If the ParameterId appears in any other Submessage, ignore the Submessage and continue with the next Submessage in the RTPS message, if any.</td>
</tr>
</tbody>
</table>

The first subspace division enables vendor-specific ParameterIds. Future minor versions of the RTPS protocol can add new parameters up to a maximum ParameterId of 0x7fff.

The range 0x8000 to 0xffff is reserved for vendor-specific options and will not be used by any future versions of the protocol.

Other specifications may reserve portions of the protocol-specific range of ParameterIds. Table 9.7 lists the ParameterIds reserved for use by other specifications and future revisions thereof. Other specifications may reserve portions of the protocol-specific range of ParameterIds. Table 9.7 lists the ParameterIds reserved for use by other specifications and future revisions thereof.

Table 9.7 - ParameterIds Reserved by other Specifications (all ranges are inclusive)

<table>
<thead>
<tr>
<th>Specification</th>
<th>Reserved ParameterIds</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS-Security 1.1</td>
<td>0x1000-0x10ff and 0x5000-0x50ff</td>
</tr>
<tr>
<td>DDS-XTypes 1.2</td>
<td>• 0x0069</td>
</tr>
<tr>
<td></td>
<td>• 0x0072-0x0075</td>
</tr>
<tr>
<td></td>
<td>• 0x3f01-0x3fff</td>
</tr>
<tr>
<td></td>
<td>• 0x7f01-0x7fff</td>
</tr>
</tbody>
</table>

For backwards compatibility, both subspaces are subdivided again. If a ParameterId is expected, but not present, the protocol will assume the default value. Similarly, if a ParameterId is present but not recognized, the protocol will either skip and ignore the parameter or treat the parameter as an incompatible QoS. The actual behavior depends on the ParameterId value, see Table 9.6.

9.4.2.12 SerializedPayload

A `SerializedPayload` SubmessageElement contains the serialized representation of either value of an application-defined data-object or the value of the key that uniquely identifies the data-object.

The specification of the process used to encode the application-level data-type into a serialized byte-stream is not strictly part of the RTPS protocol. For the purpose of interoperability, all implementations must however use a consistent representation (See, 10 Serialized Payload Representation).

The wire representation for the `SerializedPayload` is:
9.4.2.13 Count

The PSM maps the Count SubmessageElement defined in 8.3.5.10 to the structure:

```c
typedef Count_t Count;
```

Following the CDR encoding, the wire representation of the Count SubmessageElement is:

```
Count
0...2.........8.............16.............24. ............32
+------------------------------------------------------------------+
| long value |
+------------------------------------------------------------------+
```

9.4.2.14 ChangeCount

The PSM maps the ChangeCount SubmessageElement defined in 8.3.5.11 to the structure:

```c
typedef ChangeCount_t ChangeCount;
```

Following the CDR encoding, the wire representation of the ChangeCount SubmessageElement is:

```
ChangeCount
0...2.........8.............16.............24. ............32
+------------------------------------------------------------------+
| long high |
+------------+
| unsigned long low |
+----------------+
```

9.4.2.15 Checksum

The Checksum submessage element only appears as part of the HeaderExtension submessage. Depending on the value of the ChecksumFlags that appear in the HeaderExtension flags, see 9.4.5.2.1.

The PSM maps the Checksum SubmessageElement defined in 8.3.5.12 to one of three structures: `Checksum32_t`, `Checksum64_t`, or `Checksum128_t`.

This format and interpretation of the Checksum is described in Table 9.8.

Table 9.8 – Format and interpretation of the Checksum

<table>
<thead>
<tr>
<th>ChecksumFlags</th>
<th>Format</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C1, C2)</td>
<td>N/A</td>
<td>The messageChecksum is not included in the HeaderExtension</td>
</tr>
<tr>
<td>0, 0</td>
<td>Checksum32_t</td>
<td>The messageChecksum is a 32-bit checksum. It shall hold the big-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endian representation of the CRC-32C (Castagnoli) checksum of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RTPS message. The result of the CRC calculation is a 32-bit integer. It</td>
</tr>
<tr>
<td></td>
<td></td>
<td>shall be serialized into the 4-bytes of the Checksum32_t type using CDR big</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endian encoding.</td>
</tr>
<tr>
<td>1, 0</td>
<td>Checksum64_t</td>
<td>The messageChecksum is a 64-bit checksum. It shall hold the big-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endian representation of the CRC-64/XZ checksum of the RTPS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>message.</td>
</tr>
</tbody>
</table>
The full specification of the CRC checksum computation requires specifying the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,1</td>
<td>Checksum128_t The messageChecksum is a 128-bit checksum. It shall hold the MD5 digest of the RTPS message.</td>
</tr>
<tr>
<td></td>
<td>The result of the CRC calculation is a 64-bit integer. It shall be serialized into the 8-bytes of the Checksum64_t type using CDR big endian encoding.</td>
</tr>
</tbody>
</table>

Following the CDR encoding, the wire representation of the Checksum SubmessageElement is one of the following:

For ChecksumFlags (C1,C2) equal to (0,1)

```
Checksum
  0...2..........8.............16............24............32
  +---------------------------------------------------------------+
  | Checksum32_t value                                            |
  +----------------------------------------------------------------+
```

For ChecksumFlags (C1,C2) equal to (1,0)

```
Checksum
  0...2..........8.............16............24............32
  +---------------------------------------------------------------+
  | + Checksum64_t value                                          |
  | |                                                             |
  | +----------------------------------------------------------------+
```

For ChecksumFlags (C1,C2) equal to (1,1)

```
Checksum
  0...2..........8.............16............24............32
  +---------------------------------------------------------------+
  | ~ Checksum128_t value                                         |
  | |                                                             |
  | +----------------------------------------------------------------+
```

Note that as specified 8.3.7.2, the checksum shall be computed over the content of the RTPS Message, which includes the RTPS Header and HeaderExtension submessage. Moreover, for the purpose of computing the checksum, all the bytes of the messageChecksum field in the RTPS HeaderExtension shall be set to zero.

9.4.2.15.1 CRC Computation Parameters

The full specification of the CRC checksum computation requires specifying the following parameters:

- **CRC result width.** The number of bits used to encode the resulting checksum.
- **Polynomial.** The polynomial used for the CRC computation. It may be represented explicitly, or using more compact representations, such as, msbit-first (also known as the ‘normal’ representation) and lsbit-first (also known as ‘reversed’ representation).
- **Input data reflected.** Boolean value that defines whether the bits of each input byte are reflected before being processed.
- **Result data reflected.** Boolean value that defines whether the bits of the result are reflected. The result is reflected over a number of bits that correspond to the CRC result width. That is, over 32-bit for a CRC-32 and 64 bits for a CRC-64.
- **Initial value.** Integer value that defines the start condition for the CRC algorithm. The integer has the same bit size as the CRC result width. That is, 32-bits for a CRC-32 and 64 bits for a CRC-64.
- **Final XOR value.** This value is XORed at the end of the computation, resulting in the value of the checksum.

### 9.4.2.15.2 Parameters used by Checksum32_t

The parameters and the algorithm used shall correspond to the CRC-32C algorithm defined in IETF RFC 4960 Appendix B[6]. These parameters are shown in the table below:

Table 9.9 – Parameters used in the Checksum32_t computation (CRC-32C)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC result width</td>
<td>32 bits</td>
</tr>
<tr>
<td>Polynomial</td>
<td>Normal representation: 0x1EDC6F41</td>
</tr>
<tr>
<td></td>
<td>Explicit representation:</td>
</tr>
<tr>
<td></td>
<td>$x^{32} + x^{28} + x^{27} + x^{26} + x^{25} + x^{23} + x^{22} + x^{20} + x^{19} + x^{18} + x^{14} + x^{13} + x^{11} + x^{10}$</td>
</tr>
<tr>
<td></td>
<td>$+ x^{9} + x^{8} + x^{5} + 1$</td>
</tr>
<tr>
<td>Input data reflected</td>
<td>TRUE</td>
</tr>
<tr>
<td>Result data reflected</td>
<td>TRUE</td>
</tr>
<tr>
<td>Initial value</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>Final XOR value</td>
<td>0xFFFFFFFF</td>
</tr>
</tbody>
</table>

The following table illustrates the results of computing the checksum on various inputs.

Table 9.10 – Example Checksum32_t computation

<table>
<thead>
<tr>
<th>Input bytes</th>
<th>CRC-32C value</th>
<th>Checksum32_t bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 0x00 0x00 0x00</td>
<td>0x48674BC7</td>
<td>0x48 0x67 0x4B 0xC7</td>
</tr>
<tr>
<td>0xFF 0xFF 0xFF 0xFF</td>
<td>0xFFFFFFFF</td>
<td>0xFF 0xFF 0xFF 0xFF</td>
</tr>
<tr>
<td>0x33 0x22 0x55 0xAA 0xBB 0xCC 0xDD 0xEE 0xFF</td>
<td>0xB59CA09B</td>
<td>0xB5 0x9C 0xA0 0x9B</td>
</tr>
</tbody>
</table>

### 9.4.2.15.3 Parameters used by Checksum64_t

The parameters and algorithm used shall be as defined in the AUTOSAR Classic Platform release R20-11, Specification of CRC Routines, section 7.2.4 “64-bit CRC Calculation [7]. This corresponds to the CRC-64/XZ parameters shown in the table below. The polynomial used is also known as the ECMA-182 CRC-64 polynomial.

Table 9.11 – Parameters used in the Checksum64_t computation (CRC-64/XZ)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC result width</td>
<td>64 bits</td>
</tr>
<tr>
<td>Polynomial</td>
<td>Normal representation: 0x42F0E1EBA9EA3693</td>
</tr>
<tr>
<td></td>
<td>Explicit representation:</td>
</tr>
<tr>
<td></td>
<td>$x^{64} + x^{52} + x^{57} + x^{55} + x^{54} + x^{53} + x^{52} + x^{47} + x^{46} + x^{45} + x^{40}$</td>
</tr>
<tr>
<td></td>
<td>$+ x^{39} + x^{38} + x^{37} + x^{35} + x^{33} + x^{32} + x^{31} + x^{29} + x^{27} + x^{24} + x^{23} +$</td>
</tr>
<tr>
<td></td>
<td>$+ x^{22} + x^{24} + x^{19} + x^{17} + x^{13} + x^{12} + x^{10} + x^{9} + x^{7} + x^{4} + x + 1$</td>
</tr>
<tr>
<td>Input data reflected</td>
<td>TRUE</td>
</tr>
<tr>
<td>Result data reflected</td>
<td>TRUE</td>
</tr>
<tr>
<td>Initial value</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>Final XOR value</td>
<td>0xFFFFFFFF</td>
</tr>
</tbody>
</table>
The following table illustrates the results of computing the checksum on various inputs.

Table 9.12 – Example Checksum64_t computation

<table>
<thead>
<tr>
<th>Input bytes</th>
<th>CRC-64/XZ value</th>
<th>Checksum64_t bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 0x00 0x00 0x00</td>
<td>0xF4A586351E1B9F4B</td>
<td>0xF4 0xA5 0x86 0x35 0x1E 0x1B 0x9F 0x4B</td>
</tr>
<tr>
<td>0xFF 0xFF 0xFF 0xFF</td>
<td>0xFFFFFFFF00000000</td>
<td>0xFF 0xFF 0xFF 0xFF 0x00 0x00 0x00 0x00</td>
</tr>
<tr>
<td>0x33 0x22 0x55 0xAA 0xBB 0xCC 0xDD</td>
<td>0x701ECEB219A8E5D5</td>
<td>0x70 0x1E 0xCE 0xB2 0x19 0xA8 0xE5 0xD5</td>
</tr>
</tbody>
</table>

9.4.2.16 MessageLength

The PSM maps the MessageLength SubmessageElement defined in 8.3.5.13 to the structure:

```c
typedef MessageLength_t MessageLength;
```

Following the CDR encoding, the wire representation of the Length SubmessageElement is:

```
MessageLength
0...2...........8.............16...........24.............32
+---------------------------------+---------------------------------|
| unsigned long length             |                                  |
+---------------------------------+---------------------------------+
```

9.4.2.17 UExtension4

The PSM maps the Port SubmessageElement defined in 8.3.5.14 to the structure:

```c
typedef UExtension4_t UExtension4;
```

Following the CDR encoding, the wire representation of the UExtension4 SubmessageElement is:

```
UExtension8
0...2...........8.............16...........24.............32
+---------------------------------+---------------------------------|
| octet value[4]                   |                                  |
+---------------------------------+---------------------------------+
```

9.4.2.18 WExtension8

The PSM maps the Port SubmessageElement defined in 8.3.5.15 to the structure:

```c
typedef WExtension8_t WExtension8;
```

Following the CDR encoding, the wire representation of the WExtension8 SubmessageElement is:

```
WExtension8
0...2...........8.............16...........24.............32
+---------------------------------+---------------------------------|
| octet value[8]                   |                                  |
+---------------------------------+---------------------------------+
```

9.4.2.19 GroupDigest

The PSM maps the GroupDigest SubmessageElement defined in 8.3.5.10 to the structure:

```c
typedef GroupDigest_t GroupDigest;
```

Following the CDR encoding, the wire representation of the GroupDigest SubmessageElement is:
9.4.3 Additional SubmessageElements

In addition to the SubmessageElements introduced by the PIM, the UDP PSM introduces the following additional SubmessageElements.

9.4.3.1 LocatorUDPv4

The LocatorUDPv4 SubmessageElement is identical to a LocatorList SubmessageElement containing a single locator of kind LOCATOR_KIND_UDPv4. LocatorUDPv4 is introduced to provide a more compact representation when using UDP on IPv4.

Table 9.13 - Structure of the LocatorUDPv4 SubmessageElement

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>LocatorUDPv4_t</td>
<td>A single IPv4 address and port</td>
</tr>
</tbody>
</table>

The PSM maps the LocatorUDPv4 SubmessageElement to the structure:

```
typedef LocatorUDPv4_t LocatorUDPv4;
```

Following the CDR encoding, the wire representation of the LocatorUDPv4 SubmessageElement is:

LocatorUDPv4:

```
0...2...........8.............16.............24...........32
+---------------------------------------------------------------+
| octet value[4]                                                |
+---------------------------------------------------------------+
```

9.4.4 Mapping of the RTPS Header

Sub clause 8.3.8 in the PIM specifies that all messages should include a leading RTPS Header. The PSM mapping of the RTPS Header is shown below:

Header:

```
0...2...........8.............16.............24...........32
+---------------------------------------------------------------+
| 'R' | 'T' | 'P' | 'S' |
+---------------------------------------------------------------+
| ProtocolVersion version | VendorId vendorId |
+-------------------------+------------------|
|                           |                  |
| +                         +
| GuidPrefix guidPrefix    |                 |
+-------------------------+------------------|
```

The structure of the Header cannot change in this major version (2) of the protocol.

The RTPS Header includes a vendorId field, see 8.3.5.2. To be compliant with the DDS Interoperability Specification a vendor must have a reserved Vendor ID and use it. See 8.3.3.1.3 for details on where to find the current list of vendor IDs and how to request a new one to be assigned.
9.4.5 Mapping of the RTPS Submessages

9.4.5.1 Submessage Header

Sub clause 8.3.3.3 in the PIM defined the structure of all Submessages as composed of a leading SubmessageHeader followed by a variable number of SubmessageElements.

The PSM maps the SubmessageHeader into the following structure:

```c
struct SubmessageHeader {
    octet submessageId; octet flags;
    unsigned short submessageLength; /* octetsToNextHeader */
};
```

With the byte stream representation defined in 9.2.3, the submessageLength is defined as the number of octets from the start of the contents of the Submessage to the start of the next Submessage header. Given this definition, the remainder of the UDP PSM will refer to submessageLength as octetsToNextHeader. See also 9.4.5.1.3.

Following the CDR encoding, the wire representation of the SubmessageHeader is shown below:

```
SubmessageHeader: 0...2............8.............16............24............32
-----------------+-----------------+-----------------+-----------------+-----------------+
| submessageId | flags |E| ushort octetsToNextHeader |
-----------------+-----------------+-----------------+-----------------+-----------------+
|                 | following are the contents of Submessage |
-----------------+-----------------+-----------------+-----------------+-----------------+
```

This general structure cannot change in this major version (2) of the protocol. The following sub clauses discuss each member of the SubmessageHeader in more detail.

9.4.5.1.1 SubmessageId

This octet identifies the kind of Submessage. Submessages with IDs 0x00 to 0x7f (inclusive) are protocol-specific. They are defined as part of the RTPS protocol. Version 2.5 defines the following Submessages:

```c
enum SubmessageKind {
    @value(0x00) RTPS_HE,          /* HeaderExtension */
    @value(0x01) PAD,              /* Pad */
    @value(0x06) ACKNACK,          /* AckNack */
    @value(0x07) HEARTBEAT,        /* Heartbeat */
    @value(0x08) GAP,              /* Gap */
    @value(0x09) INFO_TS,          /* InfoTimestamp */
    @value(0x0c) INFO_SRC,         /* InfoSource */
    @value(0x0d) INFO_REPLY_IP4,   /* InfoReplyIp4 */
    @value(0x0e) INFO_DST,         /* InfoDestination */
    @value(0x0f) INFO_REPLY,       /* InfoReply */
    @value(0x12) NACK_FRAG,        /* NackFrag */
    @value(0x13) HEARTBEAT_FRAG,   /* HeartbeatFrag */
    @value(0x15) DATA,             /* Data */
    @value(0x16) DATA_FRAG,        /* DataFrag */
};
```

The meaning of the Submessage IDs cannot be modified in this major version (2). Additional Submessages can be added in higher minor versions. Submessages with ID's 0x80 to 0xff (inclusive) are vendor-specific; they will not be defined by future versions of the protocol. Their interpretation is dependent on the vendorId that is current when the Submessage is encountered.
9.4.5.1.1 Submessage Ranges Reserved by other Specifications

Other specifications may reserve portions of the protocol-specific range of Submessage IDs. Table 9.14 lists the Submessage IDs reserved for use by other specifications and future revisions thereof.

Table 9.14 - Submessage IDs Reserved by other Specifications (all ranges are inclusive)

<table>
<thead>
<tr>
<th>Specification</th>
<th>Reserved Submessage IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS-Security 1.1</td>
<td>0x30-0x3f</td>
</tr>
</tbody>
</table>

9.4.5.1.2 flags

Sub clause 8.3.3.3 in the PIM defines the EndiannessFlag as a flag present in all Submessages that indicates the endianness used to encode the Submessage. The PSM maps the EndiannessFlag flag into the least-significant bit (LSB) of the flags. This bit is therefore always present in all Submessages and represents the endianness used to encode the information in the Submessage. The EndiannessFlag is represented with the literal ‘E’. E=0 means big-endian, E=1 means little-endian.

The value of the EndiannessFlag can be obtained from the expression:

\[ E = \text{SubmessageHeader.flags} \& 0x01 \]

Other bits in the flags have interpretations that depend on the type of Submessage.

In the following descriptions of the Submessages, the character ‘X’ is used to indicate a flag that is unused in version 2.5 of the protocol. Implementations of RTPS version 2.5 should set these to zero when sending and ignore these when receiving. Higher minor versions of the protocol can use these flags.

9.4.5.1.3 octetsToNextHeader

The representation of this field is a CDR unsigned short (ushort).

In case octetsToNextHeader > 0, it is the number of octets from the first octet of the contents of the Submessage until the first octet of the header of the next Submessage (in case the Submessage is not the last Submessage in the Message) OR it is the number of octets remaining in the Message (in case the Submessage is the last Submessage in the Message). An interpreter of the Message can distinguish these two cases as it knows the total length of the Message.

In case octetsToNextHeader==0 and the kind of Submessage is NOT PAD or INFO_TS, the Submessage is the last Submessage in the Message and extends up to the end of the Message. This makes it possible to send Submessages larger than 64k (the size that can be stored in the octetsToNextHeader field), provided they are the last Submessage in the Message.

In case the octetsToNextHeader==0 and the kind of Submessage is PAD or INFO_TS, the next Submessage header starts immediately after the current Submessage header OR the PAD or INFO_TS is the last Submessage in the Message.

9.4.5.2 HeaderExtension Submessage

Sub clause 8.3.7 in the PIM defines the logical contents of HeaderExtension Submessage. The PSM maps the HeaderExtension Submessage to the following wire representation:
Flags in the Submessage Header

In addition to the EndiannessFlag, the HeaderExtension Submessage introduces the LengthFlag, TimestampFlag, UExtension4Flag, WExtension8Flag, ChecksumFlags and ParametersFlag. See 8.3.7.2.

The LengthFlag is represented with the literal ‘L’. L=1 means the HeaderExtension includes the messageLength.

The value of the LengthFlag can be obtained from the expression:
\[ L = \text{SubmessageHeader.flags} \& 0x02 \]

The TimestampFlag is represented with the literal ‘T’. T=1 means the HeaderExtension includes the Timestamp submessage element.

The value of the UExtension4Flag can be obtained from the expression:
\[ U = \text{SubmessageHeader.flags} \& 0x04 \]

The UExtension4Flag is represented with the literal ‘U’. U=1 means the HeaderExtension includes the uExtension4 submessage element.

The value of the UExtension4Flag can be obtained from the expression:
\[ U = \text{SubmessageHeader.flags} \& 0x08 \]

The WExtension8Flag is represented with the literal ‘W’. W=1 means the HeaderExtension includes the wExtension8 submessage element.

The value of the WExtension8Flag can be obtained from the expression:
\[ W = \text{SubmessageHeader.flags} \& 0x10 \]

The ChecksumFlags are represented with the literal ‘C’. There are three two flags: C1 and C2. When the two ‘C’ flags are set to zero, the HeaderExtension does not include the messageChecksum, any other value of the flags indicates the messageChecksum is included.

The value of the C1 and C2 flags can be obtained from the expressions:
\[ C1 = \text{SubmessageHeader.flags} \& 0x40 \]
\[ C2 = \text{SubmessageHeader.flags} \& 0x20 \]
The ParametersFlag is represented with the literal ‘P’. P=1 means the HeaderExtension includes the parameters.

The value of the ParametersFlag can be obtained from the expression:

\[ P = \text{SubmessageHeader.flags} \& 0x80 \]

### 9.4.5.3 AckNack Submessage

Sub clause 8.3.8.1 in the PIM defines the logical contents of the AckNack Submessage. The PSM maps the AckNack Submessage into the following wire representation:

\[
\begin{array}{c}
0...2 ............7 ............15 ............23 ............31 \\
+..................+..................+..................+..................+ \\
| \text{ACKNACK } |X|X|X|X|X|X|F|E| \text{octetsToNextHeader} |
| +-------------------------+-------------------------+ \\
| \text{EntityId} \text{readerId} | 
| +-------------------------------+ \\
| \text{EntityId} \text{writerId} | 
| +-------------------------------+ \\
| \text{~ SequenceNumberSet} \text{readerSNState} | \text{~} 
| +-------------------------+ \\
| \text{Count} \text{count} | 
| +-------------------------+ 
\end{array}
\]

#### 9.4.5.3.1 Flags in the Submessage Header

In addition to the EndiannessFlag, the AckNack Submessage introduces the FinalFlag (“Content” on page 46). The PSM maps the FinalFlag flag into the 2nd least-significant bit (LSB) of the flags.

The FinalFlag is represented with the literal ‘F’. F=1 means the reader does not require a Heartbeat from the writer. F=0 means the writer must respond to the AckNack message with a Heartbeat message.

The value of the FinalFlag can be obtained from the expression:

\[ F = \text{SubmessageHeader.flags} \& 0x02 \]

### 9.4.5.4 Data Submessage

Sub clause 8.3.8.2 in the PIM defines the logical contents of the Data Submessage. The PSM maps the Data Submessage into the following wire representation:

\[
\begin{array}{c}
0...2 ............8 ............16 ............24 ............32 \\
+..................+..................+..................+..................+ \\
| \text{DATA } |X|X|X|X|X|X|D|Q|E| \text{octetsToNextHeader} |
| +-------------------------+-------------------------+ \\
| \text{Flags} \text{extraFlags} | \text{octetsToInlineQos} |
| +-------------------------+ \\
| \text{EntityId} \text{readerId} | 
| +-------------------------------+ \\
| \text{EntityId} \text{writerId} | 
| +-------------------------+ \\
| + \text{SequenceNumber} \text{writerSN} |
| +-------------------------+ \\
| \text{~ ParameterList} \text{inlineQos} [only if Q==1] | \text{~} 
| +-------------------------+ \\
| \text{~ SerializedPayload} \text{serializedPayload} [only if D==1 || K==1] | \text{~} 
| +-------------------------+ 
\end{array}
\]

#### 9.4.5.4.1 Flags in the Submessage Header

In addition to the EndiannessFlag, the Data Submessage introduces the InlineQosFlag, DataFlag, and Key (see 8.3.8.3.2). The PSM maps these flags as follows:
The **InlineQosFlag** is represented with the literal ‘Q.’ Q=1 means that the **Data** Submessage contains the inlineQos SubmessageElement.

The value of the **InlineQosFlag** can be obtained from the expression:

\[ Q = \text{SubmessageHeader.flags} \& 0x02 \]

The **DataFlag** is represented with the literal ‘D.’ The value of the **DataFlag** can be obtained from the expression.

\[ D = \text{SubmessageHeader.flags} \& 0x04 \]

The **KeyFlag** is represented with the literal ‘K.’ The value of the **KeyFlag** can be obtained from the expression.

\[ K = \text{SubmessageHeader.flags} \& 0x08 \]

The **DataFlag** is interpreted in combination with the **KeyFlag** as follows:

- D=0 and K=0 means that there is no `serializedPayload` SubmessageElement.
- D=1 and K=0 means that the `serializedPayload` SubmessageElement contains the serialized Data.
- D=0 and K=1 means that the `serializedPayload` SubmessageElement contains the serialized Key.
- D=1 and K=1 is an invalid combination in this version of the protocol.

The **NonStandardPayloadFlag** is represented with the literal ‘N.’ The value of the **NonStandardPayloadFlag** can be obtained from the expression.

\[ N = \text{SubmessageHeader.flags} \& 0x10 \]

### 9.4.5.4.2 extraFlags

The `extraFlags` field provides space for an additional 16 bits of flags beyond the 8 bits provided as in the submessage header. These additional bits will support evolution of the protocol without compromising backwards compatibility.

This version of the protocol should set all the bits in the `extraFlags` to zero.

### 9.4.5.4.3 octetsToInlineQos

The representation of this field is a CDR unsigned short (ushort).

The `octetsToInlineQos` field contains the number of octets starting from the first octet immediately following this field until the first octet of the inlineQos SubmessageElement. If the `inlineQos` SubmessageElement is not present (i.e., the **InlineQosFlag** is not set), then `octetsToInlineQos` contains the offset to the next field after the `inlineQos`.

Implementations of the protocol that are processing a received submessage should always use the `octetsToInlineQos` to skip any submessage header elements it does not expect or understand and continue to process the `inlineQos` SubmessageElement (or the first submessage element that follows `inlineQos` if the `inlineQos` is not present). This rule is necessary so that the receiver will be able to interoperate with senders that use future versions of the protocol which may include additional submessage headers before the `inlineQos`.

### 9.4.5.5 DataFrag Submessage

Sub clause 8.3.8.3 in the PIM defines the logical contents of the **DataFrag** Submessage. The PSM maps the **DataFrag** Submessage into the following wire representation:
In addition to the EndiannessFlag, the DataFrag Submessage introduces the KeyFlag and InlineQosFlag (see 8.3.8.1.2). The PSM maps these flags as follows:

The InlineQosFlag is represented with the literal ‘Q’. Q=1 means that the DataFrag Submessage contains the inlineQos SubmessageElement.

The value of the InlineQosFlag can be obtained from the expression:

\[ Q = \text{SubmessageHeader.flags} \& 0x02 \]

The KeyFlag is represented with the literal ‘K.’

The value of the KeyFlag can be obtained from the expression:

\[ K = \text{SubmessageHeader.flags} \& 0x04 \]

K=0 means that the serializedPayload SubmessageElement contains the serialized Data. K=1 means that the serializedPayload SubmessageElement contains the serialized Key.

The NonStandardPayloadFlag is represented with the literal ‘N.’ The value of the NonStandardPayloadFlag can be obtained from the expression.

\[ N = \text{SubmessageHeader.flags} \& 0x08 \]

9.4.5.6 Gap Submessage

Sub clause 8.3.8.4 in the PIM defines the logical contents of the Gap Submessage. The PSM maps the Gap Submessage into the following wire representation:
9.4.5.6.1  Flags in the Submessage Header

In addition to the EndiannessFlag, the Gap Submessage introduces the GroupInfoFlag (8.3.8.4.2) and the FilteredCountFlag.

The PSM maps the GroupInfoFlag flag into the 2nd least-significant bit (LSB) of the flags.

The GroupInfoFlag is represented with the literal ‘G. G=1 means the Gap also includes a gapStartGSN and a gapEndGSN.

The value of the GroupInfoFlag can be obtained from the expression:
\[ G = \text{SubmessageHeader.flags} \& 0x02 \]

The PSM maps the FilteredCountFlag flag into the 3rd least-significant bit (LSB) of the flags.

The FilteredCountFlag is represented with the literal ‘F. F=1 means the Gap also includes a filteredCount.

The value of the FilteredCountFlag can be obtained from the expression:
\[ F = \text{SubmessageHeader.flags} \& 0x04 \]

9.4.5.7  HeartBeat Submessage

Sub clause 8.3.8.6 in the PIM defines the logical contents of the HeartBeat Submessage. The PSM maps the HeartBeat Submessage into the following wire representation:
### 9.4.5.7.1 Flags in the Submessage Header

In addition to the EndiannessFlag, the HeartBeat Submessage introduces the FinalFlag, the LivelinessFlag, and the GroupInfoFlag (8.3.8.2). The PSM maps the FinalFlag flag into the 2nd least-significant bit (LSB) of the flags, the LivelinessFlag into the 3rd least-significant bit (LSB) of the flags, and the GroupInfoFlag into the 4th least-significant bit (LSB) of the flags.

The FinalFlag is represented with the literal ‘F’. F=1 means the Writer does not require a response from the Reader. F=0 means the Reader must respond to the HeartBeat message.

The value of the FinalFlag can be obtained from the expression:

\[ F = \text{SubmessageHeader.flags \& 0x02} \]

The LivelinessFlag is represented with the literal ‘L’. L=1 means the DDS DataReader associated with the RTPS Reader should refresh the ‘manual’ liveliness of the DDS DataWriter associated with the RTPS Writer of the message. The value of the LivelinessFlag can be obtained from the expression:

\[ L = \text{SubmessageHeader.flags \& 0x04} \]

The GroupInfoFlag is represented with the literal ‘G’. G=1 means the HeartBeat includes the currentGSN, firstGSN, and lastGSN. The value of the LivelinessFlag can be obtained from the expression:

\[ G = \text{SubmessageHeader.flags \& 0x08} \]

### 9.4.5.8 HeartBeatFrag Submessage

Sub clause 8.3.8.7 in the PIM defines the logical contents of the HeartBeatFrag Submessage. The PSM maps the HeartBeatFrag Submessage into the following wire representation:

```
0...2...........7...............15.............23...............31
+------------------------------------------------------------------------+
| HEARTBEAT |X|X|X|X|X|X|G|L|F|E| octetsToNextHeader |
+------------------------------------------------------------------------+
| EntityId | readerId        |
| EntityId | writerId        |
+------------------------------------------------------------------------+
| SequenceNumber | firstSN        |
+------------------------------------------------------------------------+
| SequenceNumber | lastSN        |
+------------------------------------------------------------------------+
| Count        | count           |
+------------------------------------------------------------------------+
| SequenceNumber | currentGSN  [ only if G==1 ] |
+------------------------------------------------------------------------+
| SequenceNumber | firstGSN  [ only if G==1 ] |
+------------------------------------------------------------------------+
| SequenceNumber | lastGSN  [ only if G==1 ] |
+------------------------------------------------------------------------+
```
The HeartBeatFrag Submessage introduces no other flags in addition to the EndiannessFlag.

InfoDestination Submessage

Sub clause 8.3.8.8 in the PIM defines the logical contents of the InfoDestination Submessage. The PSM maps the InfoDestination Submessage into the following wire representation:

```
| 0...2...........8................16...............24.................32 |
| +---------------------------------------------------------------+
| | HEARTBEAT_FRAG |X|X|X|X|X|X|X|E| octetsToNextHeader |
| +---------------------------------------------------------------+
| | EntityId          readerId                                    |
| +---------------------------------------------------------------+
| | EntityId          writerId                                   |
| +---------------------------------------------------------------+
| | +                                                               |
| | | SequenceNumber      writerSN                               |
| | +                                                               |
| | | FragmentNumber      lastFragmentNum                         |
| | +                                                               |
| | | Count              count                                     |
| +---------------------------------------------------------------+
```

9.4.5.1 Flags in the Submessage Header

This Submessage has no flags in addition to the EndiannessFlag.

InfoReply Submessage

Sub clause 8.3.8.9 in the PIM defines the logical contents of the InfoReply Submessage. The PSM maps the InfoReply Submessage into the following wire representation:

```
| 0...2...........8................16...............24.................32 |
| +---------------------------------------------------------------+
| | INFO_REPLY  |X|X|X|X|X|X|M|E| octetsToNextHeader |
| +---------------------------------------------------------------+
| | ~ LocatorList      unicastLocatorList                          |
| | ~ LocatorList      multicastLocatorList [ only if M==1 ]       |
| +---------------------------------------------------------------+
```

9.4.5.10.1 Flags in the Submessage Header

In addition to the EndiannessFlag, The InfoReply Submessage introduces the MulticastFlag (8.3.6.2). The PSM maps the MulticastFlag flag into the 2nd least-significant bit (LSB) of the flags.
The MulticastFlag is represented with the literal ‘M’. M=1 means the InfoReply also includes a multicastLocatorList. The value of the MulticastFlag can be obtained from the expression:

\[ M = \text{SubmessageHeader.flags} \& 0x02 \]

### 9.4.5.11 InfoSource Submessage

Sub clause 8.3.8.10 in the PIM defines the logical contents of the InfoSource Submessage. The PSM maps the InfoSource Submessage into the following wire representation:

- INFO_TS
- INFO_SRC
- Pad
- GuidPrefix
- InfoTimestamp
- MultiprotocolVersion
- VendorId
- octetsToNextHeader

### 9.4.5.11.1 Flags in the Submessage Header

This Submessage has no flags in addition to the EndiannessFlag.

### 9.4.5.12 InfoTimestamp Submessage

Sub clause 8.3.8.11 in the PIM defines the logical contents of the InfoTimestamp Submessage. The PSM maps the InfoTimestamp Submessage into the following wire representation:

- INFO_TS
- INFO_SRC
- Pad
- GuidPrefix
- InfoTimestamp
- MultiprotocolVersion
- VendorId
- octetsToNextHeader

### 9.4.5.12.1 Flags in the Submessage Header

In addition to the EndiannessFlag, The InfoTimestamp Submessage introduces the InvalidateFlag (8.3.6.2). The PSM maps the InvalidateFlag flag into the 2nd least-significant bit (LSB) of the flags.

The InvalidateFlag is represented with the literal ‘I’. I=0 means the InfoTimestamp also includes a timestamp. I=1 means subsequent Submessages should not be considered to have a valid timestamp.

The value of the InvalidateFlag can be obtained from the expression:

\[ I = \text{SubmessageHeader.flags} \& 0x02 \]

### 9.4.5.13 Pad Submessage

Sub clause 8.3.8.13 in the PIM defines the logical contents of the Pad Submessage. The PSM maps the Pad Submessage into the following wire representation:

- PAD
- octetsToNextHeader

---

**DDSI-RTPS v2.5**

174
9.4.5.13.1 Flags in the Submessage Header

This Submessage has no flags in addition to the EndiannessFlag.

9.4.5.14 NackFrag Submessage

Sub clause 8.3.8.12 in the PIM defines the logical contents of the NackFrag Submessage. The PSM maps the NackFrag Submessage into the following wire representation:

```
0...2...........8..............16.............24.............32
+---------------------------------------------+-----------------------------+
| NACK_FRAG | X|X|X|X|X|X|X|E| octetsToNextHeader |
+---------------------------------------------+-----------------------------+
| EntityId readerId                          |
+---------------------------------------------+-----------------------------+
| EntityId writerId                          |
+---------------------------------------------+-----------------------------+
| SequenceNumber writerSN                    |
+---------------------------------------------+-----------------------------+
| ~  FragmentNumberSet fragmentNumberState   ~
+---------------------------------------------+-----------------------------+
| Count count                                |
+---------------------------------------------+-----------------------------+
```

9.4.5.14.1 Flags in the Submessage Header

This Submessage has no flags in addition to the EndiannessFlag.

9.4.5.15 InfoReplyIp4 Submessage (PSM specific)

The InfoReplyIp4 Submessage is an additional Submessage introduced by the UDP PSM.

Its use and interpretation are identical to those of an InfoReply Submessage containing a single unicast and possibly a single multicast locator, both of kind LOCATOR_KIND_UDPv4. It is provided for efficiency reasons and can be used instead of the InfoReply Submessage to provide a more compact representation.

The PSM maps the InfoReplyIp4 Submessage into the following wire representation:

```
0...2...........8..............16.............24.............32
+---------------------------------------------+-----------------------------+
| INFO_REPLY_IP4| X|X|X|X|X|X|M|E | octetsToNextHeader |
+---------------------------------------------+-----------------------------+
| LocatorUDPv4 unicastLocator                |
|                                             |
+---------------------------------------------+-----------------------------+
| LocatorUDPv4 multicastLocator              |
| { only if M==1 }                            |
+---------------------------------------------+-----------------------------+
```

9.4.5.15.1 Flags in the Submessage Header

In addition to the EndiannessFlag, The InfoReplyIp4 Submessage introduces the MulticastFlag. The PSM maps the MulticastFlag flag into the 2nd least-significant bit (LSB) of the flags.

The MulticastFlag is represented with the literal ‘M’. M=1 means the InfoReplyIp4 also includes a multicastLocator.
The value of the MulticastFlag can be obtained from the expression:

\[ M = \text{SubmessageHeader.flags} \& 0x02 \]

9.5 Mapping to UDP/IP Transport Messages

When RTPS is used over UDP/IP, a Message is the contents (payload) of exactly one UDP/IP Datagram.

9.6 Mapping of the RTPS Protocol

9.6.1 ParameterId definitions in the HeaderExtension

This version of the protocol does not specify any ParameterId that may appear in the HeaderExtension.

The ParameterId space for parameters in the HeaderExtension shall be as specified in 9.4.2.11.2 Compliant implementations encountering an unrecognized ParameterId within the HeaderExtension shall either skip and the parameter or reject the entire RTPS Message, as specified in Table 9.6.

9.6.2 Default Locators

9.6.2.1 Discovery traffic

Discovery traffic is the traffic generated by the Participant and Endpoint Discovery Protocols. For the Simple Discovery Protocols (SPDP and SEDP), discovery traffic is the traffic exchanged between the built-in Endpoints.

The SPDP built-in Endpoints are configured using well-known ports (see 8.5.3.4). The UDP PSM maps these well-known ports to the port number expressions listed in Table 9.15.

Table 9.15 - Ports used by built-in Endpoints

<table>
<thead>
<tr>
<th>Discovery traffic type</th>
<th>SPDP well-known port</th>
<th>Default port number expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multicast</td>
<td>SPDP_WELL_KNOWN_MULTICAST_PORT</td>
<td>PB + DG * domainId + d0</td>
</tr>
<tr>
<td>Unicast</td>
<td>SPDP_WELL_KNOWN_UNICAST_PORT</td>
<td>PB + DG * domainId + d1 + PG * participantId</td>
</tr>
</tbody>
</table>

where
domainId = DDS Domain identifier
participantId = Participant identifier
PB, DG, d0, d1 = tunable parameters (defined below)

The domainId and participantId identifiers are used to avoid port conflicts among Participants on the same node. Each Participant on the same node and in the same domain must use a unique participantId. In the case of multicast, all Participants in the same domain share the same port number, so the participantId identifier is not used in the port number expression.

To simplify the configuration of the SPDP, participantId values ideally start at 0 and are incremented for each additional Participant on the same node and in the same domain. That way, for a given domain, Participants can announce their presence to up to N remote Participants on a given node, by announcing to port numbers on that node corresponding to participantId 0 through N-1.

The default ports used by the SEDP built-in Endpoints match those used by the SPDP. If a node chooses not to use the default ports for the SEDP, it can include the new port numbers as part of the information exchanged during the SPDP.
9.6.2.2 User traffic

User traffic is the traffic exchanged between user-defined Endpoints (i.e., non-built-in Endpoints). As such, it pertains to all the traffic that is not related to discovery. By default, user-defined Endpoints use the port number expressions listed in Table 9.16.

Table 9.16 - Ports used by user-defined Endpoints

<table>
<thead>
<tr>
<th>User traffic type</th>
<th>Default port number expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multicast</td>
<td>PB + DG * domainId + d2</td>
</tr>
<tr>
<td>Unicast</td>
<td>PB + DG * domainId + d3 + PG * participantId</td>
</tr>
</tbody>
</table>

User-defined Endpoints may choose to not use the default ports. In that case, remote Endpoints obtain the port number as part of the information exchanged during the Simple Endpoint Discovery Protocol.

9.6.2.3 Default Port Numbers

The port number expressions use the following parameters:

DG = DomainId Gain
PG = ParticipantId Gain
PB = Port Base number
d0, d1, d2, d3 = additional offsets

Implementations must expose these parameters so they can be customized by the user.

In order to enable out-of-the-box interoperability, the following default values must be used:

PB = 7400
DG = 250
PG = 2
d0 = 0
d1 = 10
d2 = 1
d3 = 11

Given UDP port numbers are limited to 64K, the above defaults enable the use of about 230 domains with up to 120 Participants per node per domain.

9.6.2.4 Default Settings for the Simple Participant Discovery Protocol

When using the SPDP, each Participant sends announcements to a pre-configured list of locators. What ports to use when configuring these locators is discussed above. This sub clause describes any remaining settings that are required to enable plug-and-play interoperability.

9.6.2.4.1 Default multicast address

In order to enable plug-and-play interoperability, the default pre-configured list of locators must include the following multicast locator (assuming UDPv4):

DefaultMulticastLocator = {LOCATOR_KIND_UDPv4, "239.255.0.1", PB + DG * domainId + d0}

All Participants must announce and listen on this multicast address.

SPDPbuiltinParticipantWriter.readerLocators CONTAINS DefaultMulticastLocator
SPDPbuiltinParticipantReader.multicastLocatorList CONTAINS DefaultMulticastLocator

9.6.2.4.2 Default announcement rate

The default rate by which SPDP periodic announcements are sent equals 30 seconds.

SPDPbuiltinParticipantWriter.resendPeriod = {30, 0};
9.6.3 Data representation for the built-in Endpoints

9.6.3.1 Data Representation for the ParticipantMessageData Built-in Endpoints

The Behavior module within the PIM (8.4) defines the DataType ParticipantMessageData. This type is the logical content of the BuiltinParticipantMessageWriter and BuiltinParticipantMessageReader built-in Endpoints.

The PSM maps the ParticipantMessageData type into the following IDL:

```idl
typedef octet OctetArray4[4];
typedef sequence<octet> OctetSeq;
struct ParticipantMessageData {
    GuidPrefix_t participantGuidPrefix;
    OctetArray4 kind;
    OctetSeq data;
};
```

The following values for the kind field are reserved by RTPS:

```c
#define PARTICIPANT_MESSAGE_DATA_KIND_UNKNOWN {0x00, 0x00, 0x00, 0x00}
#define PARTICIPANT_MESSAGE_DATA_KIND_AUTOMATIC_LIVELINESS_UPDATE {0x00, 0x00, 0x00, 0x01}
#define PARTICIPANT_MESSAGE_DATA_KIND_MANUAL_LIVELINESS_UPDATE {0x00, 0x00, 0x00, 0x02}
```

RTPS also reserves for future use all values of the kind field where the most significant bit is not set. Therefore:

```
kind.value[0] & 0x80 == 0 // reserved by RTPS
kind.value[0] & 0x80 == 1 // vendor specific kind
```

Implementations can decide the upper length of the data field but must be able to support at least 128 bytes. Following the CDR encoding, the wire representation of the ParticipantMessageData structure is:

```
0...2........8................16..................24..............32
+----------------------------------+-
| GuidPrefix_t participantGuidPrefix |
+----------------------------------+-
| octet[4] kind                     |
+----------------------------------+-
| unsigned long data.length         |
+----------------------------------+-
| octet[] data.value                |
+----------------------------------+-
```

9.6.3.2 Simple Discovery Protocol built-in Endpoints

The Discovery Module within the PIM (8.5) defines the DataTypes SPDPdiscoveredParticipantData, DiscoveredWriterData, DiscoveredReaderData, and DiscoveredTopicData. These types define the logical contents of the data sent between the RTPS built-in Endpoints.

The PSM maps these types into the following IDL:

```idl
struct SPDPdiscoveredParticipantData {
    DDS::ParticipantBuiltinTopicData ddsParticipantData;
    ParticipantProxy participantProxy;
    Duration_t leaseDuration;
};
```

```idl
struct DiscoveredWriterData {
    DDS::PublicationBuiltinTopicData ddsPublicationData;
    DDS::PublisherProxy publisherProxy;
    DDS::TopicData topicData;
};
```
WriterProxy mWriterProxy;
};

struct DiscoveredReaderData {
    DDS::SubscriptionBuiltinTopicData ddsSubscriptionData;
    ReaderProxy mReaderProxy;
    ContentFilterProperty_t contentFilter;
};

struct DiscoveredTopicData {
    DDS::TopicBuiltinTopicData ddsTopicData;
};

where each DDS built-in topic data type is defined by the DDS specification.

The discovery data is sent using standard Data Submessages. In order to allow for QoS extensibility while preserving interoperability between versions of the protocol, the wire-representation of the SerializedData within the Data Submessage uses the format of a ParameterList SubmessageElement. That is, the SerializedData contains each QoS and other information within a separate parameter identified by a ParameterId. Within each parameter, the parameter value is represented using CDR.

For example, in order to add a vendor-specific Endpoint Discovery Protocol (EDP) in the SPDPdiscoveredParticipantData, a vendor could define a vendor-specific parameterId and use it to add a new parameter to the ParameterList contained in SPDPdiscoveredParticipantData. The presence of this parameterId would denote support for the corresponding EDP. As this is a vendor-specific parameterId, other vendors’ implementations would simply ignore the parameter and the information it contains. The parameter itself would contain any additional data required by the vendor-specific EDP represented using CDR.

For optimization, implementations of the protocol shall not include a parameter in the Data submessage if it contains information that is redundant with other parameters already present in that same Data submessage. As a result of this optimization an implementation shall omit the serialization of the parameters listed in Table 9.17.

The key-only messages for the built-in topics are defined as follows. In the case of a DATA submessage containing the SPDPdiscoveredParticipantData with KeyFlag=1, the only parameterId present within the ParameterList shall be the PID_PARTICIPANT_GUID. In the case of a DATA submessage containing one of SEDPdiscoveredPublicationData, SEDPdiscoveredSubscriptionData, or SEDPdiscoveredTopicData with KeyFlag=1, the only parameterId present within the ParameterList shall be the PID_ENDPOINT_GUID.

Table 9.17 - Omitted Builtin Endpoint Parameters

<table>
<thead>
<tr>
<th>BuiltInEndpoint</th>
<th>Parameter that shall be omitted</th>
<th>Parameter where the information on the omitted parameter can be found</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPDPdiscoveredParticipantData</td>
<td>ParticipantProxy::guidPrefix</td>
<td>ParticipantBuiltinTopicData::key</td>
</tr>
<tr>
<td>DiscoveredReaderData</td>
<td>ReaderProxy::remoteReaderGuid</td>
<td>SubscriptionBuiltinTopicData::key</td>
</tr>
<tr>
<td>DiscoveredWriterData</td>
<td>WriterProxy::remoteWriterGuid</td>
<td>PublicationBuiltinTopicData::key</td>
</tr>
</tbody>
</table>

For example, an implementation of the protocol sending DATA message containing the SPDPdiscoveredParticipantData, SEDPdiscoveredPublicationData, or SEDPdiscoveredSubscriptionData shall omit the parameter that contains the guidPrefix. The implementation of the protocol in the receiver side shall derive this value from the “key” parameter which is one of the following: “ParticipantBuiltinTopicData::key”, “SubscriptionBuiltinTopicData::key”, or “PublicationBuiltinTopicData::key”.
### 9.6.3.2.1 ParameterID values

Table 9.18 lists the Entities to which each parameterID applies and its default value. Unrecognized parameterIDs shall be treated as specified in Table 9.6.

#### Table 9.18 - ParameterId Values

<table>
<thead>
<tr>
<th>Name</th>
<th>ID</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID_PAD</td>
<td>0x0000</td>
<td>N/A</td>
</tr>
<tr>
<td>PID_SENTINEL</td>
<td>0x0001</td>
<td>N/A</td>
</tr>
<tr>
<td>PID_USER_DATA</td>
<td>0x002c</td>
<td>UserDataQosPolicy</td>
</tr>
<tr>
<td>PID_TOPIC_NAME</td>
<td>0x0005</td>
<td>string&lt;256&gt;</td>
</tr>
<tr>
<td>PID_TYPE_NAME</td>
<td>0x0007</td>
<td>string&lt;256&gt;</td>
</tr>
<tr>
<td>PID_GROUP_DATA</td>
<td>0x002d</td>
<td>GroupDataQosPolicy</td>
</tr>
<tr>
<td>PID_TOPIC_DATA</td>
<td>0x002e</td>
<td>TopicDataQosPolicy</td>
</tr>
<tr>
<td>PID_DURABILITY</td>
<td>0x001d</td>
<td>DurabilityQosPolicy</td>
</tr>
<tr>
<td>PID_DURABILITY_SERVICE</td>
<td>0x001e</td>
<td>DurabilityServiceQosPolicy</td>
</tr>
<tr>
<td>PID_DEADLINE</td>
<td>0x0023</td>
<td>DeadlineQosPolicy</td>
</tr>
<tr>
<td>PID_LATENCY_BUDGET</td>
<td>0x0027</td>
<td>LatencyBudgetQosPolicy</td>
</tr>
<tr>
<td>PID_LIVELINESS</td>
<td>0x001b</td>
<td>LivelinessQosPolicy</td>
</tr>
<tr>
<td>PID_RELIABILITY</td>
<td>0x001a</td>
<td>ReliabilityQosPolicy$^3$</td>
</tr>
<tr>
<td>PID_LIFESPAN</td>
<td>0x002b</td>
<td>LifespanQosPolicy</td>
</tr>
<tr>
<td>PID_DESTINATION_ORDER</td>
<td>0x0025</td>
<td>DestinationOrderQosPolicy</td>
</tr>
<tr>
<td>PID_HISTORY</td>
<td>0x0040</td>
<td>HistoryQosPolicy</td>
</tr>
<tr>
<td>PID_RESOURCE_LIMITS</td>
<td>0x0041</td>
<td>ResourceLimitsQosPolicy</td>
</tr>
<tr>
<td>PID_OWNERSHIP</td>
<td>0x001f</td>
<td>OwnershipQosPolicy</td>
</tr>
<tr>
<td>PID_OWNERSHIP_STRENGTH</td>
<td>0x0006</td>
<td>OwnershipStrengthQosPolicy</td>
</tr>
<tr>
<td>PID_PRESENTATION</td>
<td>0x0021</td>
<td>PresentationQosPolicy</td>
</tr>
<tr>
<td>PID_PARTITION</td>
<td>0x0029</td>
<td>PartitionQosPolicy</td>
</tr>
<tr>
<td>PID_TIME_BASED_FILTER</td>
<td>0x0004</td>
<td>TimeBasedFilterQosPolicy</td>
</tr>
<tr>
<td>PID_TRANSPORT_PRIORITY</td>
<td>0x0049</td>
<td>TransportPriorityQosPolicy</td>
</tr>
<tr>
<td>PID_DOMAIN_ID</td>
<td>0x000f</td>
<td>DomainId_t</td>
</tr>
<tr>
<td>PID_DOMAIN_TAG</td>
<td>0x4014</td>
<td>string&lt;256&gt;</td>
</tr>
<tr>
<td>PID_PROTOCOL_VERSION</td>
<td>0x0015</td>
<td>ProtocolVersion_t</td>
</tr>
<tr>
<td>PID_VENDORID</td>
<td>0x0016</td>
<td>VendorId_t</td>
</tr>
<tr>
<td>PID_UNICAST_LOCATOR</td>
<td>0x002f</td>
<td>Locator_t</td>
</tr>
<tr>
<td>PID_MULTICAST_LOCATOR</td>
<td>0x0030</td>
<td>Locator_t</td>
</tr>
<tr>
<td>PID_DEFAULT_UNICAST_LOCATOR</td>
<td>0x0031</td>
<td>Locator_t</td>
</tr>
</tbody>
</table>

$^3$ The encoding of DDS::ReliabilityQoSPolicy::kind is defined by RTPS::ReliabilityKind_t (9.3.2)
### Table 9.19 - ParameterId mapping and default values

<table>
<thead>
<tr>
<th>Name</th>
<th>Used For Fields</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID_PAD</td>
<td>-</td>
<td>N/A</td>
</tr>
<tr>
<td>PID_SENTINEL</td>
<td>-</td>
<td>N/A</td>
</tr>
</tbody>
</table>
| PID_USER_DATA                      | ParticipantBuiltinTopicData::user_data  
PublicationBuiltinTopicData::user_data  
SubscriptionBuiltinTopicData::user_data | See DDS Specification.          |
| PID_TOPIC_NAME                     | TopicBuiltinTopicData::name  
PublicationBuiltinTopicData::topic_name  
SubscriptionBuiltinTopicData::topic_name | N/A                            |
| PID_TYPE_NAME                      | TopicBuiltinTopicData::type_name  
PublicationBuiltinTopicData::type_name  
SubscriptionBuiltinTopicData::type_name | N/A                            |
| PID_GROUP_DATA                     | PublicationBuiltinTopicData::group_data  
SubscriptionBuiltinTopicData::group_data | See DDS Specification.          |
| PID_TOPIC_DATA                     | TopicBuiltinTopicData::topic_data  
PublicationBuiltinTopicData::topic_data  
SubscriptionBuiltinTopicData::topic_data | See DDS Specification.          |
| PID_DURABILITY                     | TopicBuiltinTopicData::durability  
PublicationBuiltinTopicData::durability  
SubscriptionBuiltinTopicData::durability | See DDS Specification.          |
| PID_DURABILITY_SERVICE             | TopicBuiltinTopicData::durability_service  
PublicationBuiltinTopicData::durability_service | See DDS Specification.          |
<table>
<thead>
<tr>
<th>PID</th>
<th>TopicBuiltinTopicData::property</th>
<th>PublicationBuiltinTopicData::property</th>
<th>SubscriptionBuiltinTopicData::property</th>
<th>See DDS Specification.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID_DEADLINE</td>
<td>deadline</td>
<td>deadline</td>
<td>deadline</td>
<td></td>
</tr>
<tr>
<td>PID_LATENCY_BUDGET</td>
<td>latency_budget</td>
<td>latency_budget</td>
<td>latency_budget</td>
<td></td>
</tr>
<tr>
<td>PID_LIVELINESS</td>
<td>liveliness</td>
<td>liveliness</td>
<td>liveliness</td>
<td></td>
</tr>
<tr>
<td>PID_RELIABILITY</td>
<td>reliability</td>
<td>reliability</td>
<td>reliability</td>
<td></td>
</tr>
<tr>
<td>PID_LIFESPAN</td>
<td>lifespan</td>
<td>lifespan</td>
<td>lifespan</td>
<td></td>
</tr>
<tr>
<td>PID_DESTINATION_ORDER</td>
<td>destination_order</td>
<td>destination_order</td>
<td>destination_order</td>
<td></td>
</tr>
<tr>
<td>PID_HISTORY</td>
<td>history</td>
<td>history</td>
<td>history</td>
<td></td>
</tr>
<tr>
<td>PID_RESOURCE_LIMITS</td>
<td>resource_limits</td>
<td>resource_limits</td>
<td>resource_limits</td>
<td></td>
</tr>
<tr>
<td>PID.OwnerShip</td>
<td>ownership</td>
<td>ownership</td>
<td>ownership</td>
<td></td>
</tr>
<tr>
<td>PID_OWNER_ship_STRENGTH</td>
<td>ownership_strength</td>
<td>ownership_strength</td>
<td>ownership_strength</td>
<td></td>
</tr>
<tr>
<td>PID_PRESENTATION</td>
<td>presentation</td>
<td>presentation</td>
<td>presentation</td>
<td></td>
</tr>
<tr>
<td>PID_PARTITION</td>
<td>partition</td>
<td>partition</td>
<td>partition</td>
<td></td>
</tr>
<tr>
<td>PID_TIME_BASED_FILTER</td>
<td>time_based_filter</td>
<td>time_based_filter</td>
<td>time_based_filter</td>
<td></td>
</tr>
<tr>
<td>PID_DOMAIN_ID</td>
<td>domainId</td>
<td>domainId</td>
<td>domainId</td>
<td></td>
</tr>
<tr>
<td>PID_DOMAIN_TAG</td>
<td>domainTag</td>
<td>domainTag</td>
<td>“” (empty, zero-length string)</td>
<td></td>
</tr>
<tr>
<td>PID_PROTOCOL_VERSION</td>
<td>protocolVersion</td>
<td>protocolVersion</td>
<td>protocolVersion</td>
<td>N/A</td>
</tr>
<tr>
<td>PID_VENDORID</td>
<td>vendorId</td>
<td>vendorId</td>
<td>vendorId</td>
<td>N/A</td>
</tr>
<tr>
<td>PID_UNICAST_LOCATOR</td>
<td>unicastLocatorList</td>
<td>unicastLocatorList</td>
<td>unicastLocatorList</td>
<td>N/A</td>
</tr>
<tr>
<td>PID_MULTICAST_LOCATOR</td>
<td>multicastLocatorList</td>
<td>multicastLocatorList</td>
<td>multicastLocatorList</td>
<td>N/A</td>
</tr>
<tr>
<td>PID_DEFAULT_UNICAST_LOCATOR</td>
<td>defaultUnicastLocatorList</td>
<td>defaultUnicastLocatorList</td>
<td>defaultUnicastLocatorList</td>
<td>N/A</td>
</tr>
</tbody>
</table>
### 9.6.4 ParameterId Definitions used to Represent In-line QoS

The Messages module within the PIM (8.3) provides the means for the Data (8.3.8.2) and DataFrag (8.3.8.3) Submessages to include QoS policies in-line with the Submessage. The QoS policies are contained using a ParameterList.

Sub clause 8.7.2.1 defines the complete set of parameters that can appear within the inlineQos SubmessageElement. The corresponding set of parameterIds is listed in Table 9.20. Unrecognized parameterIDs shall be treated as specified in Table 9.6.

Table 9.20 - Inline QoS parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>ID</th>
<th>IDL description of the contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID_PAD</td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>PID_SENTINEL</td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>PID_TOPIC_NAME</td>
<td></td>
<td>string&lt;256&gt;</td>
</tr>
<tr>
<td>PID_DURABILITY</td>
<td></td>
<td>DurabilityQosPolicy</td>
</tr>
<tr>
<td>PID_PRESENTATION</td>
<td></td>
<td>PresentationQosPolicy</td>
</tr>
</tbody>
</table>

| PID_DEFAULT_MULTICAST_LOCATOR | ParticipantProxy::defaultMulticastLocatorList | N/A                           |
| PID_METATRAFFIC_UNICAST_LOCATOR | ParticipantProxy::metatrafficUnicastLocatorList | N/A                           |
| PID_METATRAFFIC_MULTICAST_LOCATOR | ParticipantProxy::metatrafficMulticastLocatorList | N/A                           |
| PID_EXPECTS_INLINE_QOS      | ParticipantProxy::expectsInlineQos             | FALSE                          |
| PID_PARTICIPANT_manual_Liveliness_COUNT | ParticipantProxy::manualLivelinessCount | N/A                           |
| PID_BUILTIN_ENDPOINT_SET    | ParticipantProxy::availableBuiltinEndpoints    |                                |
| PID_BUILTIN_ENDPOINT_QOS    | ParticipantProxy::builtinEndpointQos            | N/A                           |
| PID_PARTICIPANT_LEASE_DURATION | SPDPdiscoveredParticipantData::leaseDuration | {100, 0}                      |
| PID_PARTICIPANT_GUID        | ParticipantBuiltinTopicData::key                | N/A                           |
| PID_GROUP_GUID              | WriterProxy::remoteGroupGuid                    | GUID_UNKNOWN                   |
| PID_GROUP_ENTITY_ID         | WriterProxy::remoteGroupGuid.entityId           | ENTITYID_UNKNOWN              |
| PID_ENDPOINT_GUID           | TopicBuiltinTopicData::key                      | N/A                           |
| PID_CONTENT_FILTER_PROPERTY | DiscoveredReaderData::contentFilter             | N/A                           |
| PID_DATA_MAX_SIZE_SERIALIZED | WriterProxy::dataMaxSizeSerialized              | N/A                           |
The policies that can appear in-line include a subset of the DataWriter QoS policies (ParameterId defined in 9.6.3) and some additional QoS (for which a new ParameterId is defined).

The following sub clauses describe these additional QoS in more detail.

9.6.4.1 Content filter info (PID_CONTENT_FILTER_INFO)

Following the CDR encoding, the wire representation of the ContentFilterInfo_t (see 9.3.2) in-line QoS is:

---

4 RTPS protocol versions prior to 2.4 defined this as a sequence<GUID_t>. However, some vendors were sending a GUID_t instead. Therefore, when interacting with protocol versions earlier than 2.4 this parameter should be ignored unless the receiver knows the format used by the vendor that sent the InlineQos.
The `filterResult` member is encoded as a bitmap. Bit 0 (MSB) corresponds to the first filter signature, bit 1 to the second filter signature, and so on. The content filter info in-line QoS is invalid unless

\[ \text{numBitmaps} = ([\text{numSignatures}/32] + (\text{numSignatures}\%32 ? 1 : 0)) \]

The bitmap is interpreted as follows:

### Table 9.21 - Interpretation of filterResult member in content filter info in-line QoS

<table>
<thead>
<tr>
<th>bit value</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sample was filtered by the corresponding filter and did not pass.</td>
</tr>
<tr>
<td>1</td>
<td>Sample was filtered by the corresponding filter and passed.</td>
</tr>
</tbody>
</table>

A filter’s signature is calculated as the 128-bit MD5 checksum of all strings in the filter's `ContentFilterProperty_t`. More precisely, all strings are combined into the following character array:

```c
[ contentFilteredTopicName relatedTopicName filterClassName filterExpression expressionParameters[0] expressionParameters[1] ... expressionParameters[numParams - 1] ]
```

where each individual string includes its NULL termination character. The filter signature is calculated by taking the MD5 checksum of the above character sequence.

### 9.6.4.2 Coherent set (PID_COHERENT_SET)

The coherent set in-line QoS parameter uses the CDR encoding for `SequenceNumber_t`.

As defined in 8.7.5, all `Data` and `DataFrag` submessages that belong to the same coherent set must contain the coherent set in-line QoS parameter with value equal to the sequence number of the first sample in the set.
For example, assume a coherent set contains sample updates with sequence numbers 3, 4, 5 and 6 from a given \textit{Writer}. Samples in this coherent set are identified by including the coherent set in-line QoS parameter with value 3. Some example \texttt{Data} submessages that the \texttt{Writer} can use to denote the end of this coherent set are listed in Table 9.22.

Table 9.22 - Example Data Submessages to denote the end of a coherent set

<table>
<thead>
<tr>
<th>Data Submessage Elements (subset)</th>
<th>Example 1 (new coherent set)</th>
<th>Example 2 (no coherent set)</th>
<th>Example 3 (no coherent set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFlag</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>InlineQosFlag</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>writerSN</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>InlineQos (PID_COHERENT_SET)</td>
<td>7</td>
<td>SEQUENCENUMBER_UNKNOWN</td>
<td>N/A</td>
</tr>
<tr>
<td>SerializedData</td>
<td>Valid data</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

9.6.4.3 Group Coherent Set (PID\_GROUP\_COHERENT\_SET)

The group coherent set in-line QoS parameter uses the CDR encoding for SequenceNumber_t.

As defined in 8.7.6, all \texttt{Data} submessages and the first \texttt{DataFrag} submessage belonging to a sample must contain the group coherent set in-line QoS parameter with value equal to the group sequence number of the first sample in the set.

For example, assume a group coherent set contains samples with group sequence numbers 11, 12, and 13 from two \texttt{Writers}. Samples in the coherent set are identified by including coherent set in-line QoS parameters and group coherent set in-line QoS parameters, among others. Example \texttt{Data} Submessages are listed in Table 9.23.

Table 9.23 - Example Data Submessages in a GROUP coherent set

<table>
<thead>
<tr>
<th>Data Submessage Elements (subset)</th>
<th>Data Submessage 1 (Writer 1)</th>
<th>Data Submessage 2 (Writer 2)</th>
<th>Data Submessage 3 (Writer 1)</th>
<th>End Coherent Set Sample (Writer 1)</th>
<th>End Coherent Set Sample (Writer 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFlag</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>InlineQosFlag</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>writerSN</td>
<td>4</td>
<td>8</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>InlineQos (PID_GROUP_SEQ_NUM)</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>InlineQos (PID_COHERENT_SET)</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>InlineQos (PID_GROUP_COHERENT_SET)</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>InlineQos (PID_GROUP_WRITER_INFO_SET)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>MD5([Writer1Id, Writer2Id])</td>
<td>MD5([Writer1Id, Writer2Id])</td>
</tr>
<tr>
<td>SerializedData</td>
<td>Valid data</td>
<td>Valid data</td>
<td>Valid data</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
9.6.4.4 Group Sequence Number (PID_GROUP_SEQ_NUM)

The group sequence number in-line QoS parameter uses the CDR encoding for SequenceNumber_t.

As defined in 8.7.5, all Data submessages and the first DataFrag submessage sent by DataWriters belonging to a Publisher with Presentation access scope GROUP must contain the group sequence number in-line QoS parameter with value equal to the group sequence number.

9.6.4.5 Publisher Writer Info (PID_WRITER_GROUP_INFO)

The publisher writer info in-line QoS parameter uses the CDR encoding for WriterGroupInfo_t. See clause 8.7.5.

As defined in 8.7.5, for DataWriters belonging to a Publisher with Presentation access scope GROUP, the Data submessages and the first DataFrag submessage of each sample shall contain the publisher writer info in-line QoS parameter.

The End Coherent Set Data submessage (see clause 8.7.6) for those DataWriters shall also contain the publisher writer info in-line QoS parameter.

9.6.4.6 Secure Publisher Writer Info (PID_SECURE_WRITER_GROUP_INFO)

The secure publisher writer info in-line QoS is reserved for DDS Security. In the cases when it is used it shall be added anywhere that the PID_WRITER_GROUP_INFO in-line QoS is required.

9.6.4.7 Original Writer Info (PID_ORIGINAL_WRITER_INFO)

Following the CDR encoding, the wire representation of the OriginalWriterInfo_t (see 9.3.2) in-line QoS shall be:

```
OriginalWriterInfo_t:
0...2...........8............16............24............32
+------------------------------------------------------------------------+
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
```

The original writer info parameter may appear in the Data or in the DataFrag submessages.

9.6.4.8 KeyHash (PID_KEY_HASH)

The key hash inline parameter contains the CDR encoding of the KeyHash_t. The KeyHash_t is defined as a 16-Byte octet array (see 9.3.2) therefore the key hash inline parameter just copies those 16 Bytes.

Given an Aggregated type "Foo" and an object “FooObject” of type “Foo”, the KeyHash_t computation for FooObject shall use the following algorithm:

Step 1. Define a new type “FooKeyHolder” as follows:
• Start with FooKeyHolder being defined the same way as the original Foo type, except that the FooKeyHolder shall have extensibility kind ‘FINAL’ (see DDS-XTYPES 1.3 clause 7.2.3).

• If there are any key members, then remove the non-key members from FooKeyHolder. Otherwise, do not remove any members.

• Reorder the members in ascending order of their memberId values.

Step 2. Define a new object “FooKeyHolderObject” of type FooKeyHolder. Initialize the values of FooKeyHolderObject from the FooObject, by setting the members present in FooKeyHolderObject to the same values as the corresponding members in FooObject.

Step 3. Apply steps 1 and 2 recursively to the members of FooKeyHolder if they are themselves Aggregated types (i.e., structure or union types).

Step 4. Compute the PLAIN_CDR2 Big Endian Serialization (see DDS-XTYPES 1.3 clause 7.4.2) of FooKeyHolderObject. The serialization shall be performed on a buffer that is initially aligned to the maximum alignment in PLAIN_CDR2 (i.e., 4). Furthermore, any padding bytes added due to alignment rules shall be set to zero.

Step 5.1 If the FooKeyHolder has a maximum serialized size that is less than or equal to 16 bytes, then then the KeyHash of FooObject shall be set to the result of Step 4, extended to 16 bytes. Padding bytes, if required to fill 16 bytes, shall be added at the end and set to zero.

Step 5.2 If the FooKeyHolder has a maximum serialized size that is greater than 16 bytes, then the KeyHash_t of FooObject shall be set to the MD5 Hash of the serialized bytes obtained from Step 4.

Note that according to the definition of the PLAIN_CDR2 serialization (see DDS-XTYPES 1.3 clause 7.4.2), the serialized bytes obtained in step 4 do not include any encapsulation header, type header, or member headers and use a maximum alignment of 4.

Example 1: Assume the type ”Foo” defined by the IDL shown below:

```idl
@final
struct Foo {
  @key long id;
  long x;
  long y;
};
```

Assume FooObject is an object of type Foo where the id member has been set to 0x12345678 the x member to 10 and the y field to 20.

In this case FooKeyHolder is defined as:

```idl
@final
struct FooKeyHolder {
  @key long id;
};
```

And FooKeyHolderObject is an object of type FooKeyHolder with its id member set to 0x12345678.

The result of step 4 (PLAIN_CDR2 big endian serialization) is the 4-byte stream containing the bytes

```
{ 0x12, 0x34, 0x56, 0x78 }
```

The maximum serialized size of FooKeyHolder is 4 bytes so step 5.1 applies. Therefore, the KeyHash_t is the 16-octet array:

```
{ 0x12, 0x34, 0x56, 0x78,
  0x00, 0x00, 0x00, 0x00,
```
Note that the added bytes needed to fill the 16 byte KeyHash_t array are set to zero.

**Example 2:** Assume the type "Foo" defined by the IDL shown below:

```idl
struct Foo {
  @key string<12> label;
  @key long long id;
  long x;
  long y;
};
```

Assume FooObject is an object of type Foo where the label member has been set to "BLUE" the id member has been set to 0x123456789abcdef0, the x member to 10 and the y member to 20.

In this case FooKeyHolder is defined as:

```idl
@final
struct FooKeyHolder {
  @key string<12> label;
  @key long long id;
};
```

And FooKeyHolderObject is an object of type FooKeyHolder with its label member set to "BLUE" and id set to 0x123456789abcdef0.

The result of step 4 (PLAIN_CDR2 big endian serialization) is the 20-byte stream containing the bytes:

```
{ 0x00, 0x00, 0x00, 0x05,
 0x42, 0x4c, 0x55, 0xd9,
 0x12, 0x34, 0x56, 0x78,
 0x9a, 0xbc, 0xde, 0xf0 }
```

Note that the serialization of the id member is aligned to a 4-byte boundary (as specified in PLAIN_CDR2) and the padding bytes introduced ahead of the serialized id have been set to zero.

The maximum serialized size of FooKeyHolder is 28 bytes: The serialization of the label member can take up to 17 bytes (4-byte length, 12 bytes the string contains the maximum 12 characters, and one extra byte for the terminating NUL). Serializing the id member after a maximum length string would require 11 more bytes (3 bytes of padding to get to a 4-byte alignment plus 8 bytes for the long long).

Given the maximum serialized size of FooKeyHolder, step 5.2 applies. Therefore, the KeyHash_t is obtained by computing an MD5 hash on the serialized stream from step 4, resulting in the 16-octet array:

```
{ 0xf9, 0x1a, 0x59, 0xe3,
 0x2e, 0x45, 0x35, 0xd9,
 0xa6, 0x9c, 0xd5, 0xd9,
 0xf5, 0xb6, 0xe3, 0x6e }
```

**Example 3:** Assume the type "Foo" defined by the IDL shown below:

```idl
@mutable
struct Nested {
  @key long m_long;
  long u;
  long w;
};
```

```idl
@mutable
struct Foo {
  @id(40) @key string<12> label;
};
```
@id(30) @key Nested m_nested;
@id(20) long x;
@id(10) long y;
};

Assume FooObject is an object of type Foo where the label member has been set to "BLUE", the m_nested member has been set to \( m_{\text{nested}}.m_{\text{long}} = 0x12345678 \), \( m_{\text{nested}}.u = 10 \) and \( m_{\text{nested}}.w = 20 \). Finally, the \( x \) and \( y \) members have been set to 100 and 200, respectively.

In this case FooKeyHolder is defined as:
@final
struct NestedKeyHolder {
  @key long m_long;
};

@final
struct FooKeyHolder {
  @key @id(30) NestedKeyHolder m_nested;
  @key @id(40) string<12> label;
};

Note that the members of FooKeyHolder (and NestedKeyHolder) have been reordered by their memberId.

Step 2 sets the FooKeyHolderObject object of type FooKeyHolder to have its member label set to "BLUE" and \( m_{\text{nested}}.m_{\text{long}} = 0x12345678 \).

The result of step 4 (PLAIN_CDR2 big endian serialization) is the 13-byte stream containing the bytes:

\[
\{ 0x12, 0x34, 0x56, 0x78 \\
 0x00, 0x00, 0x00, 0x05, \\
 0x42, 0x4c, 0x55, 0x45, \\
 0x00 
\}

The maximum serialized size of FooKeyHolder is 21 bytes: The serialization of the \( m_{\text{nested}} \) member takes 4 bytes and the \( \text{label} \) member can take up to 17 bytes (4-byte length, 12 bytes the string contains the maximum 12 characters, and one extra byte for the terminating NUL).

Given the maximum serialized size of FooKeyHolder, step 5.2 applies. Therefore, the KeyHash_t is obtained by computing an MD5 hash on the serialized stream from step 4, resulting in the 16-octet array:

\[
\{ 0x37, 0x4b, 0x96, 0xe2, \\
 0xe7, 0x27, 0x23, 0x7f, \\
 0x01, 0x6c, 0xc4, 0xce, \\
 0xbb, 0xe6, 0xb7, 0xe1 \}
\]

9.6.4.9 StatusInfo_t (PID_STATUS_INFO)

The status info parameter contains the CDR encoding of the StatusInfo_t. The StatusInfo_t is defined as a 4-Byte octet array (see 9.3.2) therefore the status info inline parameter just copies those 4 Bytes.

The status info parameter may appear in the Data or in the DataFrag submessages.

The StatusInfo_t shall be interpreted as a 32-bit worth of flags with the layout shown below:

\[
\begin{align*}
0 & \ldots 2 & \ldots 8 & \ldots 16 & \ldots 24 & \ldots 32 \\
\hline
\text{F} & \text{U} & \text{D} & \text{P} & \text{I} & \text{D} \\
\hline
\end{align*}
\]
The flags represented with the literal ‘X’ are unused by this version of the protocol and should be set to zero by the writer and not interpreted by the reader so that they may be used in future versions of the protocol without breaking interoperability.

The flags in the status info provide information on the status of the data-object to which the submessage refers. Specifically, the status info is used to communicate changes to the LifecycleState of a data-object instance.

The current version of the protocol defines the **DisposedFlag**, the **UnregisteredFlag**, the **FilteredFlag**.

The **DisposedFlag** is represented with the literal ‘D.’

D=1 indicates that the DDS DataWriter has disposed the instance of the data-object whose Key appears in the submessage.

The **UnregisteredFlag** is represented with the literal ‘U.’

U=1 indicates that the DDS DataWriter has unregistered the instance of the data-object whose Key appears in the submessage.

The **FilteredFlag** is represented with the literal ‘F.’

F=1 indicates that the DDS DataWriter has written as sample for the instance of the data-object whose Key appears in the submessage but the sample did not pass the content filter specified by the DDS DataReader.

If both DisposedFlag==0 and UnregisteredFlag=0, then the data-object whose Key appears in the Submessage has InstanceState ALIVE in the DDS DataWriter. In this case the value of the FilteredFlag indicates whether the sample that was written for that data-object instance passed the reader-specified filter: FilteredFlag==0 indicates the sample passed the filter and FilteredFlag==1 indicates it did not pass the filter.

Note that the protocol does not require that the DDS DataWriter propagates the "register" operation. Therefore, the DDS DataWriter can implement 'register' as a local operation. Since the DDS DataWriter register operation does not provide a data value propagating the register operation would be of limited use to theDataReader.

### 9.6.5 ParameterIds reserved for other DDS specifications

Other specifications may reserve ParameterIds. Table 9.24 below lists the ParameterIds reserved for use by other specifications and future revisions thereof.

**Table 9.24 - ParameterIds Reserved by other Specifications**

<table>
<thead>
<tr>
<th>Specification</th>
<th>Reserved ParameterId</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS-XTypes 1.1</td>
<td>0x0069, 0x0072, 0x0073, 0x0074, 0x0075, 0x0076</td>
</tr>
<tr>
<td>(clauses 7.6.2.1.3, 7.6.2.2, 7.6.2.3.3, and Annex D)</td>
<td>Note that 0x0076 was deprecated in DDS-XTypes 1.2</td>
</tr>
<tr>
<td>DDS-XTypes 1.2</td>
<td>0x0100 to 0x1FFF</td>
</tr>
<tr>
<td>(clauses 7.6.2.1.3, 7.6.2.2.2, 7.6.2.4.2, 7.6.2.4.3, and Annex D)</td>
<td>Parameter IDs in the range 0x1000 to 0x1FFF</td>
</tr>
<tr>
<td>DDS-XTypes 1.3</td>
<td>0x0080, 0x0081, 0x0082, 0x0083</td>
</tr>
<tr>
<td>(clauses 7.6.3.1.3, 7.6.3.2.2, 7.6.3.4.3, and Annex D)</td>
<td>Parameter IDs in the range 0x5000 to 0x5FFF</td>
</tr>
<tr>
<td>DDS-Security 1.1</td>
<td></td>
</tr>
<tr>
<td>(clause 7.4.1.3)</td>
<td></td>
</tr>
<tr>
<td>DDS-RPC 1.0</td>
<td></td>
</tr>
<tr>
<td>(clauses 7.6.2.1.1, 7.6.2.1.2, and 7.6.2)</td>
<td></td>
</tr>
</tbody>
</table>

### 9.6.6 ParameterIds Deprecated by the Protocol

The ParameterIds shown in Table 9.25 have been deprecated by the versions indicated in the table. These parameters should not be used by versions of the protocol equal or newer than the deprecated version unless
they are used with the same meaning as in versions prior to the deprecated version. Implementations that wish to interoperate with earlier versions should send and process the parameters in Table 9.23.

Table 9.25 – Deprecated ParameterId Values

<table>
<thead>
<tr>
<th>Name</th>
<th>ID</th>
<th>Deprecated By Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID_PERSISTENCE</td>
<td>0x0003</td>
<td>2.2</td>
</tr>
<tr>
<td>PID_TYPE_CHECKSUM</td>
<td>0x0008</td>
<td>2.2</td>
</tr>
<tr>
<td>PID_TYPE2_NAME</td>
<td>0x0009</td>
<td>2.2</td>
</tr>
<tr>
<td>PID_TYPE2_CHECKSUM</td>
<td>0x000a</td>
<td>2.2</td>
</tr>
<tr>
<td>PID_EXPECTS_ACK</td>
<td>0x0010</td>
<td>2.2</td>
</tr>
<tr>
<td>PID_MANAGER_KEY</td>
<td>0x0012</td>
<td>2.2</td>
</tr>
<tr>
<td>PID_SEND_QUEUE_SIZE</td>
<td>0x0013</td>
<td>2.2</td>
</tr>
<tr>
<td>PID_RELIABILITY_ENABLED</td>
<td>0x0014</td>
<td>2.2</td>
</tr>
<tr>
<td>PID_VARGAPS_SEQUENCE_NUMBER_LAST</td>
<td>0x0017</td>
<td>2.2</td>
</tr>
<tr>
<td>PID_RECV_QUEUE_SIZE</td>
<td>0x0018</td>
<td>2.2</td>
</tr>
<tr>
<td>PID_RELIABILITY_OFFERED</td>
<td>0x0019</td>
<td>2.2</td>
</tr>
<tr>
<td>PID_MULTICAST_IPADDRESS</td>
<td>0x0011</td>
<td>2.4</td>
</tr>
<tr>
<td>PID_DEFAULT_UNICAST_IPADDRESS</td>
<td>0x000c</td>
<td>2.4</td>
</tr>
<tr>
<td>PID_DEFAULT_UNICAST_PORT</td>
<td>0x000e</td>
<td>2.4</td>
</tr>
<tr>
<td>PID_METATRAFFIC_UNICAST_IPADDRESS</td>
<td>0x0045</td>
<td>2.4</td>
</tr>
<tr>
<td>PID_METATRAFFIC_UNICAST_PORT</td>
<td>0x000d</td>
<td>2.4</td>
</tr>
<tr>
<td>PID_METATRAFFIC_MULTICAST_IPADDRESS</td>
<td>0x000b</td>
<td>2.4</td>
</tr>
<tr>
<td>PID_METATRAFFIC_MULTICAST_PORT</td>
<td>0x0046</td>
<td>2.4</td>
</tr>
<tr>
<td>PID_PARTICIPANT_BUILTIN_ENDPOINTS</td>
<td>0x0044</td>
<td>2.4</td>
</tr>
<tr>
<td>PID_PARTICIPANT_ENTITYID</td>
<td>0x0051</td>
<td>2.4</td>
</tr>
<tr>
<td>PID_GROUP_ENTITYID</td>
<td>0x0053</td>
<td>Deprecated only in version 2.4. Valid in versions 2.0 to 2.3, 2.5 beyond.</td>
</tr>
</tbody>
</table>
10  Serialized Payload Representation

10.1  Introduction

The RTPS protocol transfers serialized application data in the SerializedPayload submessage element, see 9.4.2.12. The representation of the serialized application data is not part of the RTPS protocol. The RTPS protocol does not interpret the content of the SerializedPayload. It delivers them as an opaque set of bytes. It is the responsibility of the connectivity layer above the RTPS protocol to serialize and deserialize the application data objects into and from the SerializedPayload.

However, to detect configuration errors, the RTPS protocol provides a mechanism to ensure that the RTPS Writer and Reader have a common understanding of the format used to represent the data in the SerializedPayload. This is defined in Section 10.2.

In the case of DDS using RTPS the responsibility to serialize and deserialize the application data objects into and from the SerializedPayload rests with the DDS DataWriter and DataReader, respectively. In this situation, the content and format of the SerializedPayload is defined in sections 10.3 to 10.5.

10.2  SerializedPayloadHeader and Representation Identifier

All SerializedPayload shall start with the SerializedPayloadHeader defined below. The header provides information about the representation of the data that follows.

typedef octet RepresentationIdentifier[2];
typedef octet RepresentationOptions[2];

struct SerializedPayloadHeader {
    RepresentationIdentifier representation_identifier;
    RepresentationOptions representation_options;
};

TheSerializedPayloadHeader occupies the first four octets of the SerializedPayload as shown below:

0...2...........8.............16.............24.............32
+-------------------------------------------------------------------+
| representation_identifier | representation_options | 
+-------------------------------------------------------------------+
~ ... Bytes of data representation using a format that ... ~
~ ... depends on the RepresentationIdentifier and options ... ~
~ +-------------------------------------------------------------------+

The RepresentationIdentifier is used to identify the data representation used. The RepresentationOptions shall be interpreted in the context of the RepresentationIdentifier, such that each RepresentationIdentifier may define the representation_options that it requires.

For alignment purposes, the CDR stream is logically reset at the position that follows the representation_options. Therefore, there should be no initial padding before the serialized data is added to the CDR stream5.

---

5 Versions of the RTPS specification previous to version 2.4 did not clearly state where the CDR stream was reset for alignment purposes. Therefore implementations may need to take into account the vendor and protocol version when interpreting the Serialized Data.
10.3 SerializedPayload for RTPS discovery built-in endpoints

The SerializedPayload for the data messages associated with built-in discovery endpoints shall use the RepresentationIdentifier values and formats defined in Table 10.1 below.

The current version of the protocol does not use the representation options: The sender shall set the representation_options to zero. The receiver shall ignore the value of the representation_options.

Table 10.1 - RepresentationIdentifier values for built-in endpoints

<table>
<thead>
<tr>
<th>Representation Identifier</th>
<th>Value</th>
<th>Representation Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL_CDR_BE</td>
<td>0x00, 0x02</td>
<td>ParameterList (9.4.2.11). Both the parameter list and its parameters are encapsulated using OMG CDR Big Endian. See also DDS-XTypes clause 7.4.1 (Extended CDR Representation, encoding version 1) and 7.4.1.2 (Parameterized CDR Representation).</td>
</tr>
<tr>
<td>PL_CDR_LE</td>
<td>0x00, 0x03</td>
<td>ParameterList (9.4.2.11). Both the parameter list and its parameters are encapsulated using OMG CDR Little Endian. See also DDS-XTypes clause 7.4.1 (Extended CDR Representation, encoding version 1) and 7.4.1.2 (Parameterized CDR Representation).</td>
</tr>
</tbody>
</table>

10.4 SerializedPayload for other RTPS built-in endpoints

The SerializedPayload for the data messages associated with built-in endpoints other than discovery built-in endpoints shall use one of the RepresentationIdentifier values and formats defined in Table 10.2 below.

Table 10.2 - RepresentationIdentifier values for built-in endpoints other than discovery

<table>
<thead>
<tr>
<th>Representation Identifier</th>
<th>Value</th>
<th>Representation Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDR_BE</td>
<td>0x00, 0x00</td>
<td>Classic CDR representation with Big Endian encoding. See DDS-XTypes clause 7.4.1.1.</td>
</tr>
<tr>
<td>CDR_LE</td>
<td>0x00, 0x01</td>
<td>Classic CDR representation with Little Endian encoding. See DDS-XTypes clause 7.4.1.1.</td>
</tr>
<tr>
<td>PL_CDR_BE</td>
<td>0x00, 0x02</td>
<td>ParameterList (9.4.2.11) with Big Endian encoding. See also DDS-XTypes clause 7.4.1.2.</td>
</tr>
<tr>
<td>PL_CDR_LE</td>
<td>0x00, 0x03</td>
<td>ParameterList (9.4.2.11) with Little Endian encoding. See also DDS-XTypes clause 7.4.1.2.</td>
</tr>
</tbody>
</table>

The definition of each of those builtin Endpoints should indicate the serialized data format and RepresentationIdentifier used.

10.5 SerializedPayload for user-defined DDS Topics

The SerializedPayload for the data messages associated with the user-defined DDS Topics shall use the data representations defined in DDS-XTYPES clause 7.4 (Data Representation). Accordingly, the RepresentationIdentifier values and the corresponding formats shall be as defined in Table 10.3.
Table 10.3 - RepresentationIdentifier values for user-defined topic data

<table>
<thead>
<tr>
<th>RepresentationIdentifier (see DDS-XTYPES Table 60)</th>
<th>Value</th>
<th>Representation Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDR_BE</td>
<td>{0x00, 0x00}</td>
<td>Classic CDR representation with Big Endian encoding. See DDS-XTypes [3] clause 7.4.1.1.</td>
</tr>
<tr>
<td>CDR_LE</td>
<td>{0x00, 0x01}</td>
<td>Classic CDR representation with Little Endian encoding. See DDS-XTypes [3] clause 7.4.1.1.</td>
</tr>
<tr>
<td>PL_CDR_BE</td>
<td>{0x00, 0x02}</td>
<td>ParameterList (9.4.2.11) with Big Endian encoding. See also DDS-XTypes [3] clause 7.4.1.2.</td>
</tr>
<tr>
<td>PL_CDR_LE</td>
<td>{0x00, 0x03}</td>
<td>ParameterList (9.4.2.11) with Little Endian encoding. See also DDS-XTypes [3] clause 7.4.1.2.</td>
</tr>
<tr>
<td>CDR2_BE</td>
<td>{0x00, 0x10}</td>
<td>Plain CDR representation (version2) with Big Endian encoding. Similar to Classic CDR except it uses a maximum alignment of 4 bytes. See DDS-XTypes [3] clause 7.4.2 and 7.4.3.</td>
</tr>
<tr>
<td>CDR2_LE</td>
<td>{0x00, 0x11}</td>
<td>Plain CDR representation (version2) with Little Endian encoding. Similar to Classic CDR except it uses a maximum alignment of 4 bytes. See DDS-XTypes [3] clause 7.4.2 and 7.4.3.</td>
</tr>
<tr>
<td>PL_CDR2_BE</td>
<td>{0x00, 0x12}</td>
<td>Extended CDR representation (version2) for MUTABLE types with Big Endian encoding. A generalization of ParameterList. See DDS-XTypes [3] clause 7.4.2 and 7.4.3.</td>
</tr>
<tr>
<td>PL_CDR2_LE</td>
<td>{0x00, 0x13}</td>
<td>Extended CDR representation (version2) for MUTABLE types with Little Endian encoding. A generalization of ParameterList. See DDS-XTypes [3] clause 7.4.2 and 7.4.3.</td>
</tr>
<tr>
<td>D_CDR_BE</td>
<td>{0x00, 0x14}</td>
<td>Extended CDR representation (version2) for APPENDABLE types with Big Endian encoding. Similar to plain CDR2_BE except for a delimiter. See DDS-XTypes [3] clause 7.4.2 and 7.4.3.</td>
</tr>
<tr>
<td>D_CDR_LE</td>
<td>{0x00, 0x15}</td>
<td>Extended CDR representation (version2) for APPENDABLE types with Little Endian encoding. Similar to plain CDR2_BE except for a delimiter. See DDS-XTypes [3] clause 7.4.2 and 7.4.3.</td>
</tr>
<tr>
<td>XML</td>
<td>{0x00, 0x04}</td>
<td>See DDS-XTypes [3] clause 7.4.4.</td>
</tr>
</tbody>
</table>

Legacy DDS implementations that are not compliant with DDS-XTYPES should minimally support the RepresentationIdentifier values CDR_BE and CDR_LE and the type system elements specified in clause F1 (Type System) in Annex F (Characterizing Legacy DDS Implementations) of the DDS-XTYPES specification.

10.6 Example for Built-in Endpoint Data

Following is the SerializedPayload element used by the SEDPbuiltinSubscriptionsWriter to declare a DataReader.

The DataReader is for Topic “Square” and type “ShapeType”. The DataReader has the Endpoint GUID c0:a8:02:05:00:00:3a:20:00:00:00:02:80:00:00:07, DESTINATION_ORDER kind
The corresponding SerializedPayload element has the following layout:

```
<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>identifier = PL_LE</td>
<td>options = 0x0000</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>paramId = PID_ENDPOINT_GUID</td>
<td>parameterLength = 16</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>value of the GUID (16 Bytes)</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>paramId = PID_TOPIC_NAME</td>
<td>parameterLength = 12</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDR_Serialization(&quot;Square&quot;).length = 7</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'S'</td>
<td>'q'</td>
<td>'u'</td>
<td>'a'</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r'</td>
<td>e'</td>
<td>'\0'</td>
<td>padding</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>paramId = PID_TYPE_NAME</td>
<td>parameterLength = 16</td>
<td>44</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDR_Serialization(&quot;ShapeType&quot;).length = 10</td>
<td>48</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'S'</td>
<td>'h'</td>
<td>'a'</td>
<td>'p'</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e'</td>
<td>'T'</td>
<td>'y'</td>
<td>'p'</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e'</td>
<td>'\0'</td>
<td>padding</td>
<td>padding</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>paramId = PID_DESTINATION_ORDER</td>
<td>parameterLength = 4</td>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDR_Serialization(kind = BY_SOURCE_TIMESTAMP) = 1</td>
<td>68</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PID_DEADLINE</td>
<td>parameterLength = 8</td>
<td>72</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDR_Serialization(deadline.second) = 3</td>
<td>76</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDR_Serialization(deadline.fraction) = 0</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pId = PID_SENTINEL</td>
<td>parameterLength = 0</td>
<td>84</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The representation identifier is PL_LE, indicating little Endian representation.

The BY_SOURCE_TIMESTAMP, and DEADLINE period of 3 seconds. The remaining members have their default values, so they are not serialized into the SerializedPayload.
The actual bytes of the `SerializedPayload` element are shown below:

```
| 0x00 | 0x03 | 0x00 | 0x00 | 4  |
| 0x5A | 0x00 | 0x10 | 0x00 | 8  |
+-----+-----+-----+-----+-----+
| 0xC0 | 0xA8 | 0x02 | 0x05 | 12 |
+-----+-----+-----+-----+-----+
| 0x00 | 0x00 | 0x34a| 0x20 | 16 |
+-----+-----+-----+-----+-----+
| 0x00 | 0x00 | 0x00 | 0x02 | 20 |
+-----+-----+-----+-----+-----+
| 0x80 | 0x00 | 0x00 | 0x07 | 24 |
+-----+-----+-----+-----+-----+
| 0x05 | 0x00 | 0x0C | 0x00 | 28 |
+-----+-----+-----+-----+-----+
| 0x07 | 0x00 | 0x00 | 0x00 | 32 |
+-----+-----+-----+-----+-----+
| 0x53 | 0x71 | 0x75 | 0x61 | 36 |
+-----+-----+-----+-----+-----+
| 0x72 | 0x65 | 0x00 | padding | 40 |
+-----+-----+-----+-----+-----+
| 0x07 | 0x00 | 0x10 | 0x00 | 44 |
+-----+-----+-----+-----+-----+
| 0x0A | 0x00 | 0x00 | 0x00 | 48 |
+-----+-----+-----+-----+-----+
| 0x53 | 0x68 | 0x61 | 0x70 | 52 |
+-----+-----+-----+-----+-----+
| 0x65 | 0x54 | 0x79 | 0x70 | 56 |
+-----+-----+-----+-----+-----+
| 0x65 | 0x00 | 0x00 | 0x00 | 60 |
+-----+-----+-----+-----+-----+
| 0x25 | 0x04 | 0x00 | 0x00 | 64 |
+-----+-----+-----+-----+-----+
| 0x01 | 0x00 | 0x00 | 0x00 | 68 |
+-----+-----+-----+-----+-----+
| 0x23 | 0x08 | 0x00 | 0x00 | 76 |
+-----+-----+-----+-----+-----+
| 0x03 | 0x00 | 0x00 | 0x00 | 76 |
+-----+-----+-----+-----+-----+
| 0x00 | 0x00 | 0x00 | 0x00 | 80 |
+-----+-----+-----+-----+-----+
| 0x01 | 0x00 | 0x00 | 0x00 | 84 |
+-----+-----+-----+-----+-----+
```

10.7 Example for User-defined Topic Data

Following is the `SerializedPayload` element used by an application DataWriter to send Data on the Topic “Square” with type “ShapeType” defined by the IDL below. The DataWriter uses PLAIN_CDR representation with encoding version 1 and Little Endian byte order.

```c
@final
struct ShapeType {
    @key string<64> color;
    long x;
    long y;
    long size;
};
```

The representation identifier is CDR_LE.
The example uses a data value with color set to “BLUE”, x = 34, y = 100, size = 24
The corresponding `SerializedPayload` element has the following layout:

```
<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier = CDR_LE</td>
<td>options = 0x0000</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDR_Serialization(&quot;BLUE&quot;).length = 5</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'B'</td>
<td>'L'</td>
<td>'U'</td>
<td>'E'</td>
<td>12</td>
</tr>
<tr>
<td>\0</td>
<td>padding</td>
<td>padding</td>
<td>padding</td>
<td>16</td>
</tr>
<tr>
<td>CDR_Serialization(x) = 34</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDR_Serialization(y) = 100</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDR_Serialization(size) = 24</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The actual bytes of the `SerializedPayload` element are shown below:

```
<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>options = 0x0000</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>0x05</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>8</td>
</tr>
<tr>
<td>0x42</td>
<td>0x4c</td>
<td>0x55</td>
<td>0x45</td>
<td>12</td>
</tr>
<tr>
<td>0x00</td>
<td>padding</td>
<td>padding</td>
<td>padding</td>
<td>16</td>
</tr>
<tr>
<td>0x22</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>20</td>
</tr>
<tr>
<td>0x64</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>24</td>
</tr>
<tr>
<td>0x18</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>28</td>
</tr>
</tbody>
</table>
```
A References
