Date: March 2018

DDS for eXtremely Resource Constrained Environments

Revised Submission

Real-Time Innovations, Inc.

Twin Oaks Computing, Inc.

eProsima, Inc.

OMG Document Number: mars/2018-03-03

Associated Files (Normative):

dds-xrce_types.idl (mars/2018-03-08)

dds-xrce_model.xmi (mars/2018-03-05)

Associated Files (Non-normative):

dds-xrce_model.eap (mars/2018-03-06)

IPR mode: *Non-Assert*

Submission Contacts:

Gerardo Pardo-Castellote, Ph.D. (lead) CTO,
Real-Time Innovations, Inc. gerardo AT rti.com

Clark Tucker,
CEO, TwinOaks Computing, Inc. ctucker AT twinoakscomputing.com

Jaime Martin-Losa
CTO, eProsima. JaimeMartin AT eprosima.com

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

RESTRICTED RIGHTS LEGEND

TRADEMARKS

COMPLIANCE

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 http://www.omg.org, under Documents, Report a Bug/Issue (http://www.omg.org/report_issue.)

Table of Contents

Preface

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 http://www.omg.org/.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Specifications are available from the OMG website at:

http://www.omg.org/spec

Specifications are organized by the following categories:

Business Modeling Specifications

Middleware Specifications

- **CORBA/IIOP**
- **Data Distribution Services**
- **Specialized CORBA**

IDL/Language Mapping Specifications

Modeling and Metadata Specifications

- **UML, MOF, CWM, XMI**
- **UML Profile**

Modernization Specifications

Platform Independent Model (PIM), Platform Specific Model (PSM), Interface Specifications

- **CORBAServices**
- **CORBAFacilities**

CORBA Embedded Intelligence Specifications

CORBA Security Specifications

OMG Domain Specifications

Signal and Image Processing Specifications

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
109 Highland Avenue
Needham, MA 02494
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 section 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.

`Courier - 10 pt. Bold:` Programming language 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 http://www.omg.org/report_issue.htm.

0 Response Details

0.1 OMG Response Details

This specification is a response to the OMG RFP "eXtremely Resource Constrained Environments DDS (DDS-XRCE)" (mars/2016-03-21).

0.2 Copyright Waiver

0.3 Contacts

- Gerardo Pardo-Castellote, Ph.D. (lead) CTO, Real-Time Innovations, Inc. gerardo AT rti.com
- Clark Tucker, CEO, TwinOaks Computing, Inc. ctucker AT twinoakscomputing.com
- Jaime Martin-Losa, CTO, eProsima. JaimeMartin AT eprosima.com

0.4 Problem Statement

DDS offers a rich set of API and QoS policies to develop distributed applications for highly complex, mission critical systems; however, these features come at relatively high cost in terms of memory, CPU and bandwidth usage. A large number of devices on the edge of the network, such as actuators and sensors, do not need these types of features. Being at the edge of the network, it is sufficient enough for these devices to be able to publish and subscribe to data while at the same time appearing as first-class participants in the DDS Global Data Space.

While the DDS programming API can be avoided by an application, it is still necessary to implement the DDS-RTPS wire protocol to be able to participate in DDS communication. However, the RTPS protocol, whilst featuring a wire efficiency comparable to that of MQTT and other IoT standards, is designed for situations where:

- The underlying network transport has reasonably large MTUs exceeding those available in LoWPANs, such as 802.15.4 and ZigBee.
- The protocol endpoints are assumed to have continuous and simultaneous presence on the network, rather than operating on independent sleep/wake cycles.

This peer-to-peer and broker-less nature of DDS has significant advantages in many situations, but it also has drawbacks in extremely resource-constrained environments (XRCEs). Specifically, the complexity of a DDS peer and the discovery and presence traffic are generally greater in DDS than in client-broker-based technologies, making it harder for DDS implementations to be extremely small.

The purpose of this specification is to enable resource-constrained devices to participate in DDS communication, while at the same time allowing those devices to be disconnected for long periods of time but still be discoverable by other DDS applications.

0.5 Overview of this Specification

This specification defines a protocol (the XRCE Protocol) that allows resource-constrained devices (XRCE Clients) to participate as first-class DDS participants in the DDS Global Data Space.

The XRCE Client uses the XRCE Protocol to interact with a service (the XRCE Agent). The XRCE Agent acts on behalf of the device and makes the device discoverable in the larger DDS network while isolating the device from the complexity of DDS itself

The behavior of the XRCE Agent is described in terms of an XRCE Object Model and a set of operations to manipulate the object model. The XRCE Agent operations are triggered by messages sent from the XRCE Client to the Agent using the XRCE Protocol. As a result of these operations the XRCE Agent can also send messages back to the client.

The XRCE Object Model acts as a façade to the Standard DDS Object model. Each XRCE Object is associated with one or more DDS Objects in that operations on XRCE Objects cause operations to be executed in related DDS Objects and vice versa. The operations on DDS Objects behave as specified by the DDS (and DDS-RTPS) specifications. In this way the interaction of XRCE Clients with regular DDS DomainParticipants is fully specified.

The specification is organized into the following sections:

- XRCE Object Model (clause 7). Specifies the XRCE Object models. The operations on each object and how they related to the corresponding DDS objects.
- XRCE Protocol (clause 8). Specifies the structure and content of the messages used by the XRCE protocol (8.3), their protocol interactions (8.4) and how they relate to the operations on the XRCE Objects (8.5).
- XRCE Agent Configuration (clause 9). Specifies a portable way to configure the XRCE Agent.
- XRCE Deployments (clause 10). Specifies deployment scenarios and typical use-cases:
    - XRCE Client to DDS DomainParticipant communication
    - XRCE Client to Client communication brokered by the XRCE Agent
    - XRCE Client to Client direct (peer-to-peer) communication
- Transport Mappings (clause 11). Maps the protocol to various transports such as UCP and TCP.
- IDL Types (annex A). Defines all the types used in the specification using IDL.
- Example Messages (Non-Normative) (annex B). Provides examples for all XRCE message types with explanations.

0.6 Statement of Proof of Concept

The submitters have created a working implementation of the protocol. There is a canned demonstration of the implementation available to interested parties. This demo has been delivered to the MARS task force at the New Orleans Meeting (September 2017).

This implementation proved the following points.

- The implementations is very simple can be executed in very resource constrained targets. It has been demonstrated on a microcontroller with 256 KB of RAM using the NuttX operating system and we saw no reason it could not scale to much smaller environments.
- It took one engineer less than 1 months and it uses less than 2000 line of code.
- The implementation fully interoperates with multiple DDS implementations and tools. These perceive the XRCE client as a regular DDS DomainParticipant.
- The implementation is robust to sleeping cycles and transport-level disconnects.

0.7 Mapping to RFP Requirements

0.7.1 Mandatory requirements

Table 1 lists the Mandatory Requirements in the RFP and how the submission addresses all of them.

Table 1 – Mandatory Requirements

Requirement Number	Description	How is the requirement addressed
6.5.1	*Submitters shall define a DDS-XRCE protocol between a Client and an Agent.*	The submission meets this requirement. Defined in clause 8 (XRCE Protocol).
6.5.2	*The DDS-XRCE protocol shall allow DDS-XRCE clients to be perceived by DDS Participants as at least minimum profile DDS Participants.*	The submission meets this requirement. Defined in clause 7 (XRCE Object Model), clause 8 (XRCE Protocol), and clause 10 (XRCE Deployments). The DDS-XRCE Client is associated with XRCE Client within the agent which contains one or more XRCE DomainParticipant (see Figure 5) each XRCE DomainParticipant has an associated DDS::DomainParticipant that is instantiated in the Agent. Other DDS Participants perceive the client as the DDS::DomainParticipant(s) instantiated in the Agent on behalf of that client. DDS::DomainParticipant(s) instantiated in the Agent may use a full DDS implementation so it can appear as a DDS Participants that meets the minimum profile of any other DDS profile.
6.5.3	*DDS-XRCE protocol overhead, when sending or receiving user data, shall not exceed 24 bytes per packet.*	The submission meets this requirement. Typical overhead is 12 bytes and 16 bytes as worst case. See discussion in 0.8.1
6.5.4	*Submissions shall address clients operating with sleep/wakeup periods.*	The submission meets this requirement. The use of proxy objects in the Agent (section 7.5) protocol lets the Agent keep the state of the DDS-XRCE Client and its presence in the network even if it goes into sleep cycles. The use of a protocol where the client can send its *sessionId* and *clientKey* on each message (section 8.3.2) allows the logical "connection/session" between the client and the Agent to survive network-level disconnects/timeouts that may occur while the client is in long sleep cycle.
6.5.5	*Submissions shall define the DDS-XRCE protocol for the UDP/IP transport.*	The submission meets the requirement. Defined in clause 11 (Transport Mappings).

6.5.6	*Submissions shall define the DDS-XRCE protocol for the TCP/IP transport.*	The submission meets the requirement. Defined in clause 11 (Transport Mappings).

0.7.2 Non-Mandatory requirements

Table 2 lists the Non-Mandatory Requirements in the RFP and how the submission addresses all of them.

Table 2 – Non-Mandatory Requirements

Requirement Number	Description	How is the requirement addressed
6.6.1	*Submitters may consider defining support for transparent user payload compression*	This can be done by the application by compressing the data before sending to the DDS-XRCE and after receiving it from the DDS-XRCE. Since the DDS-XRCE standard does not define an application API, a vendor can easily offer this or a user may wrap the vendor API to do it.
6.6.2	*Submitters may consider defining a custom mapping to IEEE 802.15.4 that does not rely on IP.*	The protocol transport model (section 11.1) makes the DDS-XRCE protocol independent of the transport. While the submission does not define a custom protocol to mapping to IEEE 802.15.4 nothing in the specification precludes it.
6.6.3	*Submissions may define the use of the TCP/IP transport mapping over HTML5 WebSockets.*	Similar to 6.6.2. The submission does not explicitly define this but there is nothing that precludes such a deployment.

0.7.3 Internalization Support

This specification is not affected by internalization issues. The DDS-XRCE protocol does not use strings, and the strings that may appear in the object and data representations leverage the existing XCDR and XML mechanisms to handle character sets.

0.8 Responses to the RFP Issues to be discussed

0.8.1 Submissions shall clearly quantify the protocol overhead

There are multiple possible definitions of "protocol overhead". In our opinion, the most relevant one for DDS-XRCE deployments is the number of non application-payload bytes that are required by the protocol to send and receive each individual data message above and beyond what the underlying network transport (e.g. TCP or Bluetooth) requires.

For example, if the client is a temperature sensor that wants to send its temperature to DDS, the data-payload may be a 2-byte integer representing the temperature in Celsius. To send these two bytes, the protocol needs to place additional bytes onto the transport (e.g. TCP or Bluetooth) message.

For messages that contain a single data element the overhead of the XRCE protocol is defined as any bytes added beyond the serialized application data. Overhead includes sequence numbers or any other session or object identifiers.

As described in sections 8.3.2, 8.3.3, 8.3.4, and 8.3.5 the overhead consists of the MessageHeader, the SubmessageHeader and any additional bytes added to the payload beyond the ones that contain the serialized application data. The resulting overhead is from 12 to 16 bytes:

- The MessageHeader can be 4 or 8 Bytes (section 8.3.2). The extra 4 bytes are needed to include the clientKey used for authentication.
- The SubmessageHeader is 4 Bytes (section 8.3.3).
- The submessages that carry application data (WRITE_DATA, DATA) define a payload that adds 4 bytes (BaseObjectRequest).

For example B.3.1 and B.5.1 show the 12-byte overhead on the messages used to send and receive data, respectively.

XRCE messages can also contain meta-data such as a timestamp or application sequence number. This can add 12 extra bytes but it is not considered overhead because they are optional and only added if the XRCE Client desire the information to be there.

The overhead of a single data element is not necessarily the only or most relevant metric. XRCE Clients often need to send or receive multiple data values and the XRCE protocol has messages that can include multiple data values. In these cases the “overhead bytes” are shared across multiple data elements and the overhead per data element is much reduced.

For example B.5.2 shows that it is possible to receive a sequence of data samples in a single message that has a 16-byte overhead. If this were used to receive 4 samples the overhead would be 4 bytes per sample. If it were 16 samples then it would be 1 byte per sample.

Finally it is important to note that often what is most relevant is the overhead relative to the overall message size, which includes the overhead of the underlying transport layers.

Using this “relative” definition and TCP as an example, the minimum overhead is 12 bytes. See the DATA example in B.5. As an absolute number compared to the very small 2-byte payload, the 12-byte overhead may seem like a lot. But when considering that TCP/IP adds 40 bytes of overhead (20 for IP and 20 for TCP), the extra 12 represent only 20% of the total packet. If the data payload was a more realistic size (e.g. 16 bytes), then the 12-byte XRCE overhead would be even less in relative terms.

While there are things that could be done to lower this overhead, it would have significant cost in terms of flexibility, overhead, and robustness. This is further discussed below. Reducing from 12 bytes to 6 bytes would take the worst-case relative overhead down from 20% to 12%. This best-case saving of 8% does not seem to warrant the aforementioned costs.

0.8.1.1 Use of 8-bit submessageId, 8-bit flags, and 16-bit sequence number

These 4 bytes could be reduced by using fewer bits for each of these fields.

Reducing the bits in the submessageId would limit the future evolution of the protocol. Likewise, reducing the bits in the flags would also limit the future evolution of the protocol.

Currently, there are 14 message types used and 8 bits used, leaving room for 256 messages. Reducing to 4 bits would only allow 16 messages, which are too close to the current number and moreover would not leave any range for vendor-specific messages. So at least 5 bits would be required. If that was done, the 3 remaining bits could be used for flags. Some messages, however, already use 4 bits of flags, so they would need to be re-designed. All in all, this optimization would only save 1 byte, which does not appear to be a worthwhile tradeoff.

Reducing the bits in the sequence number would limit the number of outstanding messages that can be "in flight," which limits the speed at which messages are sent, especially in situations with large round-trip times. Reducing the bits would also affect the number of messages that could be cached when the client is disconnected, so it may cause message loss in deep-sleep cycle scenarios. Alternatively, variable-length encoding could be used for the sequence number, although this introduces additional complexities. Also, variable-length encoding uses 8 bits for small value integers (less than 127), 16 bits for integers smaller than 16129, and 24 bits for integers between 16130 and 65535. So when averaging over all possible ranges of a 16-bit integer, the variable-length encoding uses more bytes (2.7 on average) than the straight 2-byte encoding.

0.8.1.2 Representing object as a String Name

Most DDS-XRCE Objects may be represented using a string Name, XML string, and binary XCDR. See 7.7.3.

The use of a String name provides an extremely simple and compact representation that can be used to represent very large objects. It is typically used to represent QoS profiles, DDS data-types, or even entire DDS applications. The idea is that the definition of these objects is provided to the Agent via a configuration (e.g., reading a file or using an external tool). Then a client can represent this object by just sending the (small) Object Name. That way, a few bytes sent over the wire allow a DS-XRCE Client to request a complex QoS to be used or to instantiate a DataReader or DataWriter of a very complex data-type.

This mechanism lets the DDS-XRCE client have full access to the DDS QoS settings and the full type system without significant resource use.

The CDR representation of a string encodes the length of the string as a 32-bit unsigned integer followed by the string characters, including the terminating NUL. For short (less than 255 length) strings, this encoding can have significant overhead. For example, an 8-chracter string would utilize at least 13 bytes (4 for the length, 8 for the characters, and 1 for the NUL). This encoding can be worse because the encoding of the length needs to be 4-byte aligned. In the worst case, it could introduce 3 additional bytes of padding. So overall, the serialized string could take 13 to 16 bytes—an overhead of 60% to 100%.

To reduce this overhead, the specification could add a more compact encoding for short strings, defined as those whose maximum length is less than 255. This encoding would use only 1 byte for the length and not include the trailing NUL. With this optimization, the same 8-character string would take 9 bytes. So the overhead would be reduced to 12.5%.

0.8.1.3 Representing object as a String Name, XML string

The use of an XML string provides a complete representation of any QoS, Type, or DDS-Entity. The advantage of this representation over the String Name is that it allows the client to define new QoS and Types that were not pre-configured in the Agent.

The XML string representation reuses the existing DDS-XML standard formats.

While the use of XML may appear verbose, it is only used to configure or initialize the system (e.g., define the QoS or data types), so this "one-time" cost may be worth it in many deployments in order to have the full flexibility of dynamically defining the objects. Moreover, the XML representation may not be so verbose if all it does is modify an existing definition. For example, the following XML (145 bytes) can be used to define a custom profile by extending an existing profile called "ExistingProfile" and modifying the history depth:

```
<qos_profile name="MyProf" base_name="ExistingProfile " >
    <datawriter_qos>
        <history>
            <depth>10</depth>
        </history>
    </datawriter_qos>
</qos_profile>
```

While this could be made more compact by using a different "string" representation (e.g., JSON), XML has the advantage that we can reuse the existing DDS-XML specification. Also, JSON still lacks a standard way to represent schemas. We believe that use of JSON would be a worthwhile future extension to this specification that would become

possible when the previous two issues (standard for JSON representation of DDS resources and standard JSON schema language) are addressed. Note that there are planned or existing standardization efforts both at the IETF and the OMG to address these issues.

0.8.1.4 Use of XCDR (binary) representation of resources

The use of XCDR binary representation affords a very compact way to represent objects. This representation reuses the same IDL and XCDR serialization formats already supported by DDS implementations. While it would have been possible to define even smaller binary representations of objects, doing so would have required defining custom binary formats that are currently not supported in the DDS ecosystem. Defining these custom binary formats may increase the complexity and cost of the implementation, and limit its adoption. So the reuse of existing IDL and XCDR appears to be the best tradeoff.

0.8.2 Submissions shall clearly explain supported DDS profiles

All **the DDS profiles—minimum, content, ownership, and durability**—are supported.

The submission provides complete access to all the DDS QoS and data-types. Stated differently, the other DDS participants will observe the DDS-XRCE Client as the DDS DomainParticipant proxy managed by the Agent. The DomainParticipant proxy can be configured with any QoS, and publish and subscribe any Data type.

Furthermore, the DataReader representation (see struct OBJK_DataReader_Binary in A) allows the DataReader to specify time- and content-based filters.

Consequently, the rest of the system will perceive the DDS-XRCE Client as a full DDS implementation.

1 Scope

This specification defines a XRCE Protocol between a resource constrained, low-powered device (client) and an Agent (the server). The XRCE Protocol enables the device to communicate with a DDS network and publish and subscribe to topics in a DDS domain via an intermediate service (the XRCE Agent). The specification's purpose and scope are to ensure that applications based on different vendors' implementations of the XRCE Protocol and XRCE Agent are compatible and interoperable.

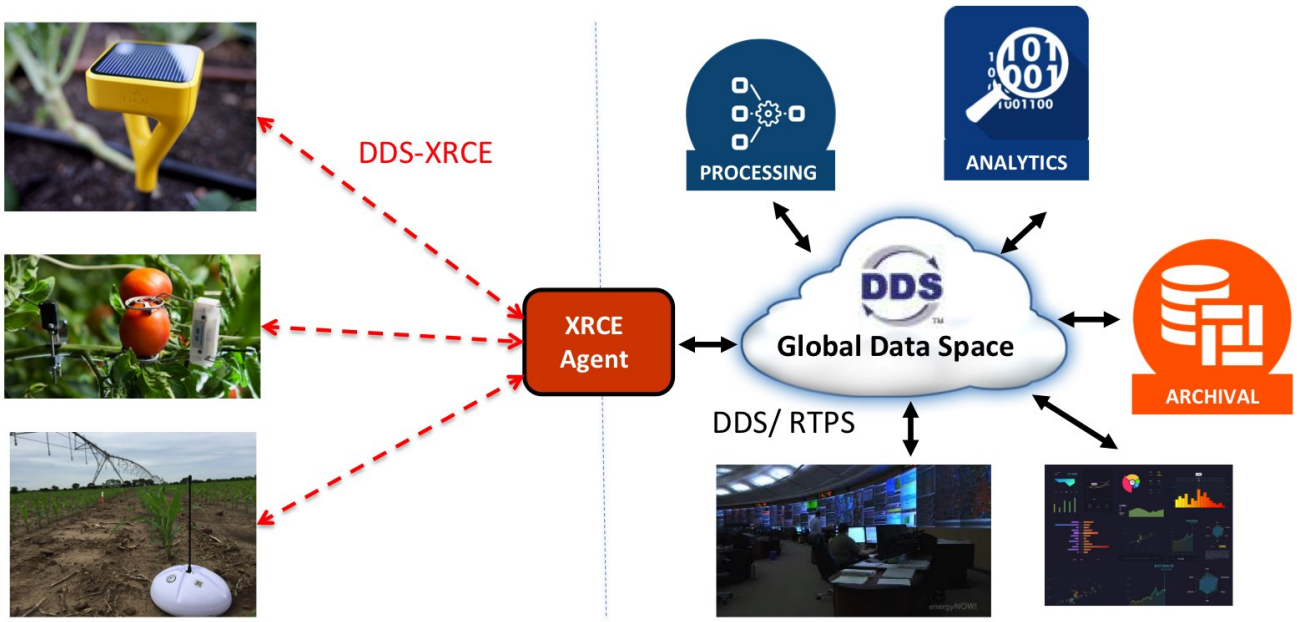


Figure 1— Scope of XRCE Protocol

The XRCE protocol is a client-server protocol between resource-constrained devices (clients) and an XRCE Agent (server). The protocol allows the resource contrained devices with sleep/wake cycles to have access to the DDS Global Data Space over limited-bandwith networks.

2 Conformance

This specification defines ten profiles. Each constitutes a separate conformance point:

- **Read Access profile**. Provides the clients the ability to read data on pre-configured Topics with pre-configured QoS policies. Requires implementation of all submessage types except for CREATE, INFO, WRITE_DATA, and DELETE, including the associated behaviors.
- **Write Access profile**. Provides the clients the ability to write data on pre-configured Topics with pre-configured QoS policies. Requires implementation of all submessage types except for CREATE, INFO, READ_DATA, DATA, and DELETE, including the associated behaviors.
- **Configure Entities profile**. Provides the clients the ability define DomainParticipant, Topic, Publisher, Subscriber, DataWriter, and DataReader entities using pre-configured QoS policies and data-types. Requires implementation of the CREATE_CLIENT, DELETE_CLIENT, CREATE, and DELETE submessage and the associated behaviors.
- **Configure QoS profile**. Provides client the ability to define QoS profiles to be used by DDS entities. Requires implementation of the CREATE submessage and the associated behaviors for object kind OBJK_QOSPROFILE.
- **Configure types profile**. Provides client the ability to explicitly define data types to be used for DDS Topics. Requires implementation of the CREATE submessage and the associated behaviors for object kind OBJK_TYPE.
- **Discovery access profile**. Provides the clients the ability to discover the Topics and Types available on the DDS Global Data Space. Requires implementation of the GET_INFO and INFO submessage and the associated behaviors.
- **File based configuration profile**. Provides a standard way to configure the Agent using XML files. Requires implementation of the file-based configuration mechanism described in clause 9.3
- **UDP Transport profile**. Implements the mapping of the protocol to the UDP transport. Requires implementing the mechanisms described in clause 11.2 (UDP Transport).
- **TCP Transport profile**. Implements the mapping of the protocol to the UDP transport. Requires implementing the mechanisms described in clause 11.3 (TCP Transport).
- **Complete profile**. Requires implementation of the complete specification.

3 References

3.1 Normative References

The following normative documents contain provisions that, 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.

- [IETF RFC-1982] Serial Number Arithmetic. https://tools.ietf.org/html/rfc1982
- [IDL] Interface Definition Language (IDL), version 4.2, http://www.omg.org/spec/IDL/
- [DDS] Data Distribution Service for Real-Time Systems Specification, version 1.4 http://www.omg.org/spec/DDS/
- [DDS-XML] DDS Consolidated XML Syntax, version 1.0, http://www.omg.org/spec/DDS-XML/
- [DDS-XTYPES] Extensible And Dynamic Topic Types for DDS, version 1.2, http://www.omg.org/spec/DDS-XTypes/
- [UML] Unified Modeling Language, version 2.5, http://www.omg.org/spec/UML/2.5
- [UDP] User Datagram Protocol, IETF RFC 768, https://tools.ietf.org/html/rfc768.
- [TCP] Transmission Control Protocol, STD 7, IETF RFC 793, https://tools.ietf.org/html/rfc793.
- [DTLS] Datagram Transport Layer Security, version 1.2, IETF RFC 6347, https://tools.ietf.org/html/rfc6347
- [TLS] The Transport Layer Security (TLS) Protocol, version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246
-
-

3.2 Non-normative References

- [SMART] Smart Transducers Specification, version 1.0, https://www.omg.org/spec/SMART/

4 Terms and Definitions

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

Data Distribution Service (DDS)

An OMG distributed data communications specification that allows Quality of Service policies to be specified for data timeliness and reliability. It is independent of implementation languages.

DDS Domain

Represents a global data space. It is a logical scope (or “address space”) for Topic and Type definitions. Each Domain is uniquely identified by an integer Domain ID. Domains are completely independent from each other. For two DDS applications to communicate with each other they must join the same DDS Domain.

DDS DomainParticipant

A DomainParticipant is the DDS Entity used by an application to join a DDS Domain. It is the first DDS Entity created by an application and serves as a factory for other DDS Entities. A DomainParticipant can join a single DDS Domain. If an application wants to join multiple DDS Domains, then it must create corresponding DDS DomainParticipant entities, one per domain**.**

DDS Global Data Space

The “DDS Global Data Space” consists of a collection of peers communicating over the Data Distribution Service and the collection of data observable by those peers.

GUID

Globally Unique Identifier

5 Symbols

Acronyms	Meaning
DDS	Data Distribution Service
IDL	Interface Definition Language
RTPS	Real-Time Publish-Subscribe
XRCE	Extremely Resource Constrained Environments

6 Additional Information

6.1 Changes to Adopted OMG Specifications

This specification does not change any adopted OMG specification.

6.2 Acknowledgements

The following companies submitted this specification:

- Real-Time Innovations, Inc.
- eProsima
- TwinOaks Computing

7 XRCE Object Model

7.1 General

This specification defines a wire protocol, the DDS-XRCE protocol, to be used between an XRCE Client and XRCE Agent. The XRCE Agent is a DDS Participant in the DDS Global Data Space. The DDS-XRCE protocol allows the client to use the XRCE Agent as a proxy in order to produce and consume data in the DDS Global Data Space.

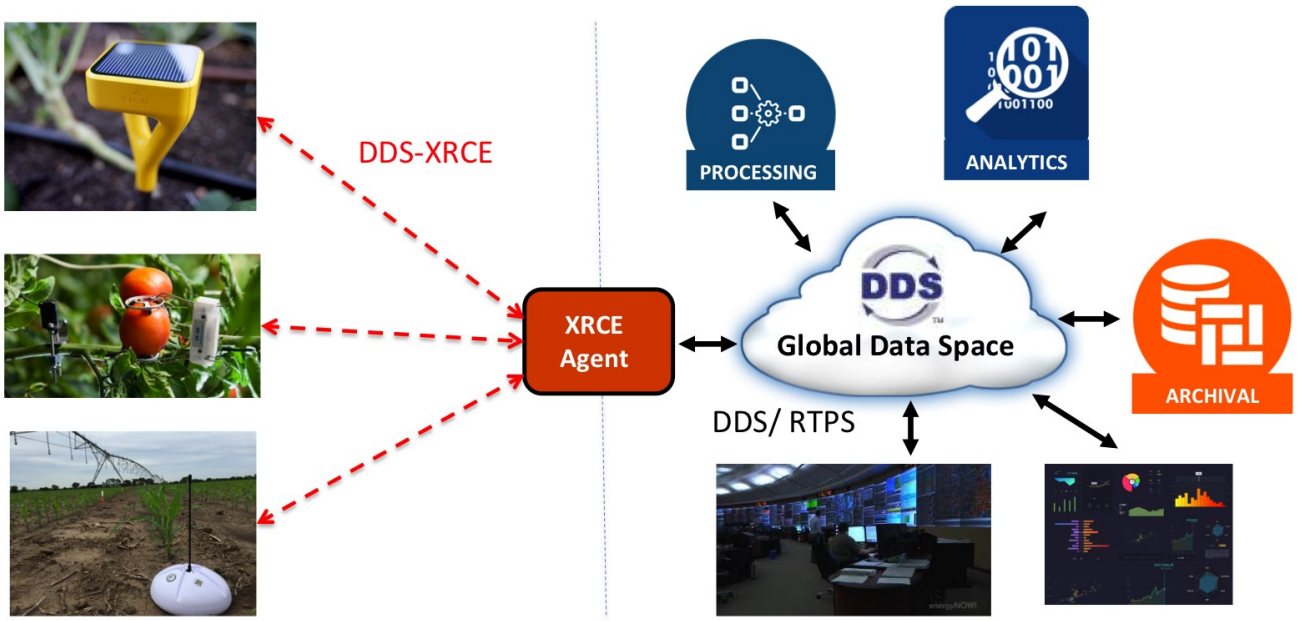


Figure 2— Scope of XRCE Protocol

The XRCE protocol is a client-server protocol between resource-constrained devices (clients) and an XRCE Agent (server). The protocol allows the resource contrained devices with sleep/wake cycles to have access to the DDS Global Data Space over limited-bandwith networks.

To model the interaction between the XRCE Client and XRCE Agent, this specification defines a UML model for the XRCE Agent. This model, called the DDS-XRCE Object Model, defines the objects, interfaces, and operations to be implemented by the agent. It also defines how they relate to operations on the Standard DDS Object Model as defined in the OMG Data-Distribution Service Specification [DDS].

Because the target environment is a resource-constrained device, the goal with the DDS-XRCE object model is not to expose the complete Standard DDS object model. It is understood that much of the configuration can be performed directly on the Agent and therefore does not require explicit interaction from the client. Instead, the focus is on the core set of features required to enable DDS-XRCE clients to participate in a meaningful way in the DDS data-space. In addition to the exposed object from the Standard DDS Object model, the DDS-XRCE object model defines new objects needed to manage disconnected clients, as well as to enable access control and access rights.

The DDS-XRCE protocol is defined as a set of logical messages exchanged between the XRCE Client and the DDS-XRCE Agent. These messages perform logical actions on the DDS-XRCE Object Model that result in corresponding actions on the Standard DDS Object Model. The specification of these logical actions fully describes the observable behavior of the XRCE Agent and its interactions both with the Client and the DDS Global Data Space.

The DDS-XRCE Object Model is similar to the Standard DDS Object Model. Compared to the DDS Object Model it is simpler having a reduced number of objects and operations. This makes the model suitable for resource-constrained, low-power clients. However it also it includes additional features that support remote clients, such as, an access control model and application management model. Despite being simpler, the DDS-XRCE Object Model provides XRCE clients complete access to the DDS Global Data space. Any DDS Topic may be published or subscribed to on any DDS with

any QoS. This is illustrated in Figure 3.

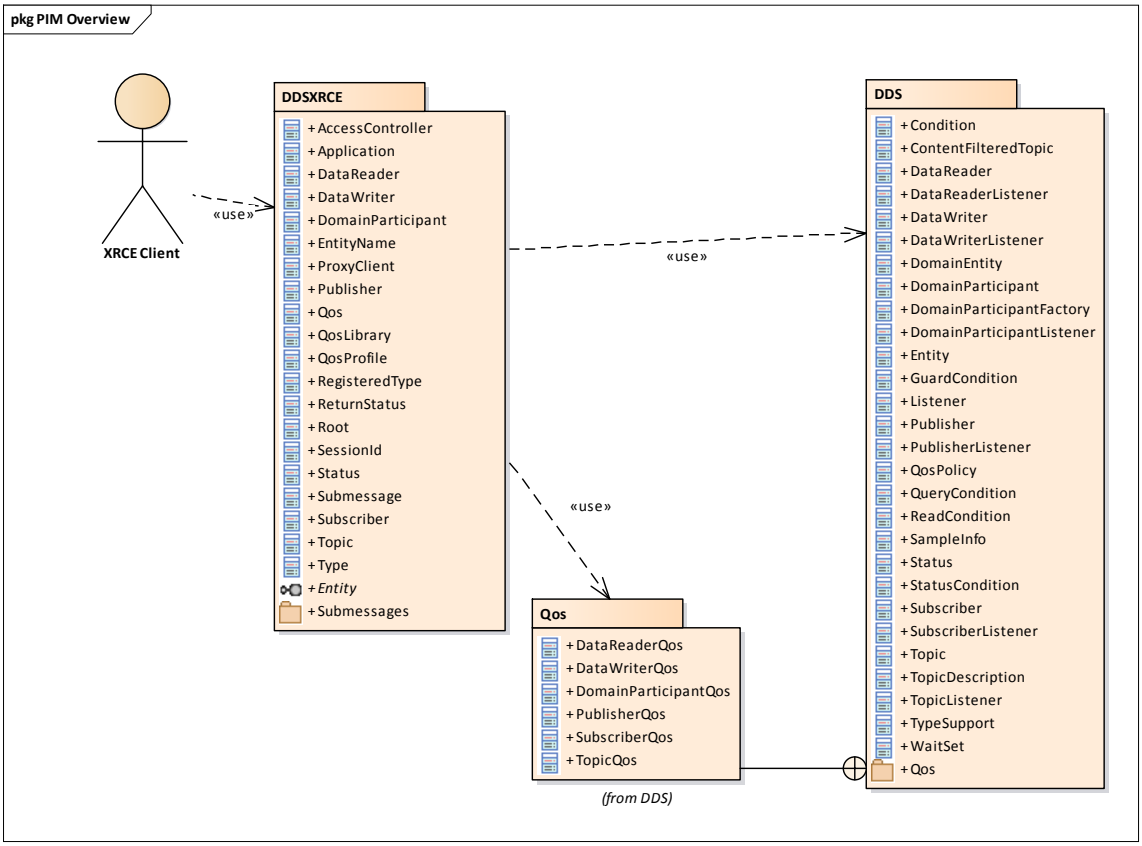


Figure 3— DDS-XRCE Object Model Overview

The DDS-XRCE Object Model is contained in the package DDS-XRCE. It acts as a façade to the Standard DDS Object Model (defined in the DDS specification. The Standard DDS Objects are shown contained in the DDS package.

7.2 XRCE Client

The DDS-XRCE Client (XRCE Client) is exposed to the DDS-XRCE Object Model and the façade object. Logically, one can think of this as equivalent to the "DDS Object Model". However, a client never interacts directly with objects in the Standard Object Model, and there is not a one-to-one mapping between the operations on the DDS-XRCE Object Model and the "DDS Object Model". This specification does not simply reuse the standard "DDS Object Model" and operations for three reasons:

1. The DDS Object Model is intended for use with a local programming API. For this reason, the DDS Object Model contains many objects and methods with strongly typed parameters, as well as a direct callback interface by means of listener objects that the application registers with the middleware. Such an API is not suitable for resource-constrained, low-power clients that typically prefer more "resource-oriented interfaces." These clients expect a simplified interface with no callbacks, and use parameters encoded in text.

2. The XRCE Client connectivity is assumed to be inherently intermittent due to potentially aggressive use of low-power mode and deep sleep to conserve battery or loss of radio connectivity. The DDS-XRCE DDS Object Model must overcome intermittent connectivity by introducing a "session," which can exist across repeated sleep-wakeup cycles by a device.

3. The XRCE Client can access a DDS Service from any location. Therefore, it is desirable to have an access control model that authenticates each client application/principal, controls whether the principal can access the DDS Global Data Space, and controls which operations each principal can perform (e.g., which DDS Topics it can publish and subscribe).

This specification recognizes that XRCE Client entities may have very different needs. Therefore, it supports clients with a wide range of requirements:

- Simple devices may not need to perform any discovery interaction with the XRCE Agent other then (a) having their presence detected by the agent, (b) establishing a presence in the DDS data-space, and (c) being able to publish data of a well-known DDS Topic using a DDS QoS policy. Such a client does not need any of the QoS configuration and dynamic entity creation capabilities of DDS.

- More capable devices may need to publish and subscribe to well-known Topics; however, an XRCE Client may not want the data to be pushed by the XRCE Agent at an arbitrary time, for example due to network constraints. Thus, the DDS model of "pushing" data from Writers to Readers may not work well. This specification addresses this constraint by enabling a device to activate/deactivate "data push" from the Agent.

- Advanced clients may choose to utilize DDS concepts and create their own XRCE Agent resources that map to DDS Objects. These clients may also want to control the Qos of the DDS Objects. This specification enables these types of Clients by exposing a set of operations to dynamically create/update/delete Agent objects. This handling of agents/clients stands in contrast to the first two cases, in which all resources are known in advance and pre-configured on the Agent.

- Finally, complex clients may need to be aware of advanced concepts, such as sequence-numbers (or sample IDs), timestamps, and DDS sources.

As shown by this list, this specification enables simple devices with little to no configuration ability to communicate with fully capable DDS devices.

7.3 XRCE Agent

The purpose of the DDS-XRCE Agent (XRCE Agent) is to *establish* and *maintain* the presence of the XRCE Client in the DDS data-space. This specification does not dictate any particular implementation; instead the required behavior is described as a set of logical operations on the DDS-XRCE Object Model.

An important feature of this specification is the simplified interaction with the XRCE Agent. The agent presents an Object Model that describes *resources*. At a high-level, a resource is an object that can be accessed with a name and has properties and behavior. Resources may be preconfigured with well-known names, or dynamically created by an XRCE Client.

Examples of named resources in the XRCE Agent are:

- XRCE Type

- XRCE DataWriter

- XRCE DataReader

Any XRCE Client that is allowed to communicate with the XRCE Agent and has the required access rights can refer to these resources by name. Thus, if an XRCE Agent is preconfigured with a resource named "MySquareWriter" that can publish a type "ShapeDemoTypes::ShapeType", a Client that has access to this resource can write data using this resource simply by referring to the existing "MySquareWriter". The Client does not have to create a resource.

Some resource implementation details are outside the scope of this specification. For example, a resource "MySquareWriter" may be associated with a DDS DataWriter shared by many DDS-XRCE clients, or an XRCE Client may have its own dedicated "MySquareWriter", as long as the DDS DataWriter supports the client's required QoS policies.

An important feature of the DDS-XRCE Object Model is a Client's ability to query the Model, as opposed to the typical behavior in the Standard DDS Object Model, in which changes are updated and pushed in real-time. That model is likely not suitable for target environments where disconnected devices are expected to be common. This specification enables Clients to be in charge of when data is received, and to request the XRCE Agent to return data that matches a set of

constraints. Thus, an XRCE Client that is disconnected will not be woken up by an XRCE Agent (it may not be possible); instead, an XRCE Client queries the XRCE Agent when it wakes up.

It is important to distinguish between the operations on the DDS-XRCE Object Model and the Standard DDS Object Model. There is not a 1-to-1 mapping between the operations. Specifically, any reference to the Standard DDS Object Model refers to the behavior and semantics defined in the DDS specification. The DDS operations on the Standard DDS Object Model are not necessarily exposed to, or have an equivalent in, the DDS-XRCE Object Model. The XRCE Agent is not required to expose any programming APIs; the standard interactions occurring with the XRCE Client use the DDS-XRCE protocol, while interactions with other DDS domain participants use the DDS-RTPS protocol.

7.4 Model Overview

At the highest level, the DDS-XRCE Object Model consists of 5 classes: The `Root` singleton, `ProxyClient`, `Application`, `AccessController`, and `DomainParticipant`.

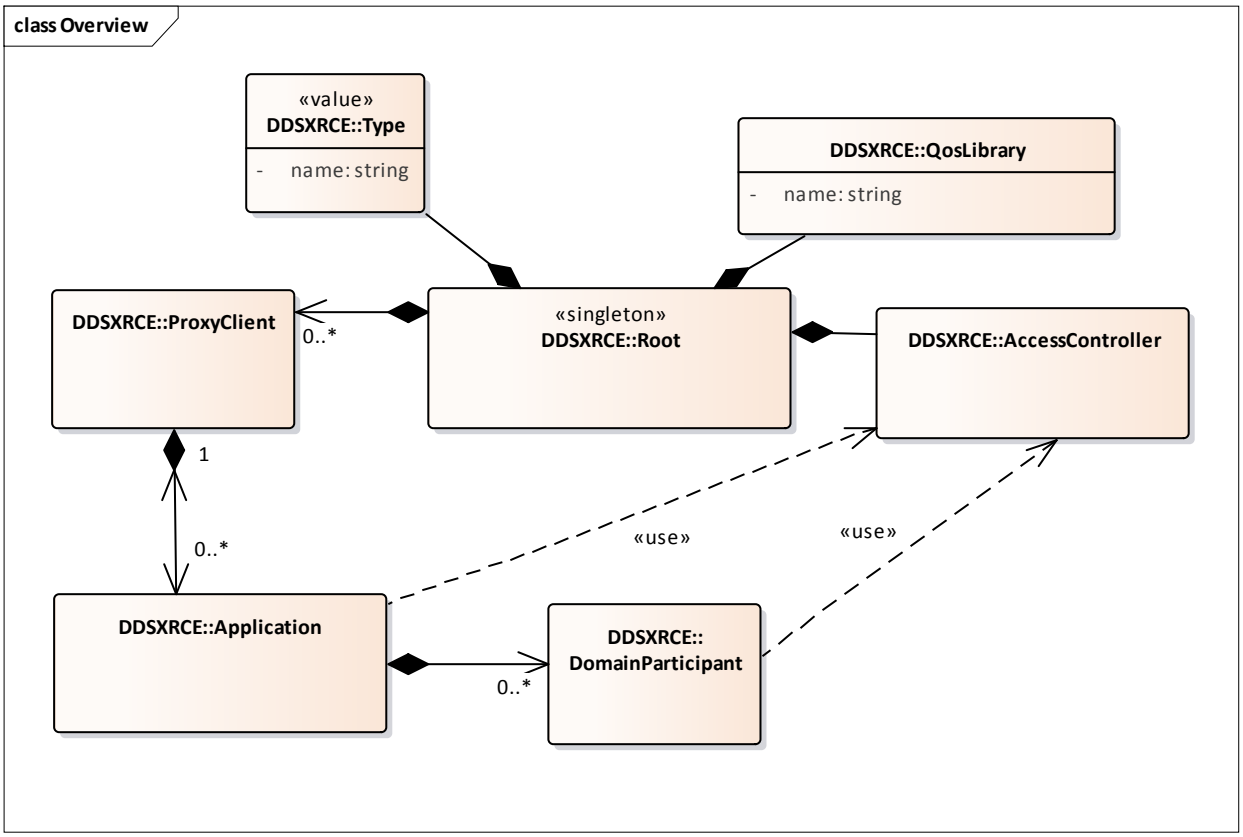


Figure 4 — DDS-XRCE Object Model Overview

The `Root` singleton is the entry point for the service. It functions as a factory for all the Objects managed by the XRCE `Agent`.

The `ProxyClient` class models the XRCE `Client` application that interacts with the XRCE `Agent` using the XRCE protocol. Each `Application` object is associated with a single XRCE `ProxyClient` and gets its access rights from those assigned to the XRCE `Client`.

The `Application` class models a software application that connects with the XRCE `Client` and manages the DDS objects needed to publish and subscribe data on one or more DDS `Domains`. An XRCE `Application` can be associated with zero of more `DomainParticipant` objects.

The `AccessController` is responsible for making decisions regarding the resources and operations a particular XRCE `ProxyClient` is allowed to perform. It contains rules that associate a `Client` with privileges, which determine which DDS domain an application executing on behalf of a client may join, which DDS `Topics` the application can read and write, and so on.

The DDS-XRCE `DomainParticipant` is a proxy for a DDS `DomainParticipant` and models the association with a DDS domain and the capability of the Application to publish and subscribe to Topics on that domain.

7.5 XRCE DDS Proxy Objects

Several of the DDS-XRCE objects act as proxies to corresponding DDS objects. These proxy objects allow the client application to participate as first-class citizens on the DDS network by delegating the actual DDS behavior and DDS-RTPS protocol implementation to the proxy DDS objects.

This relationship is shown in Figure 5.

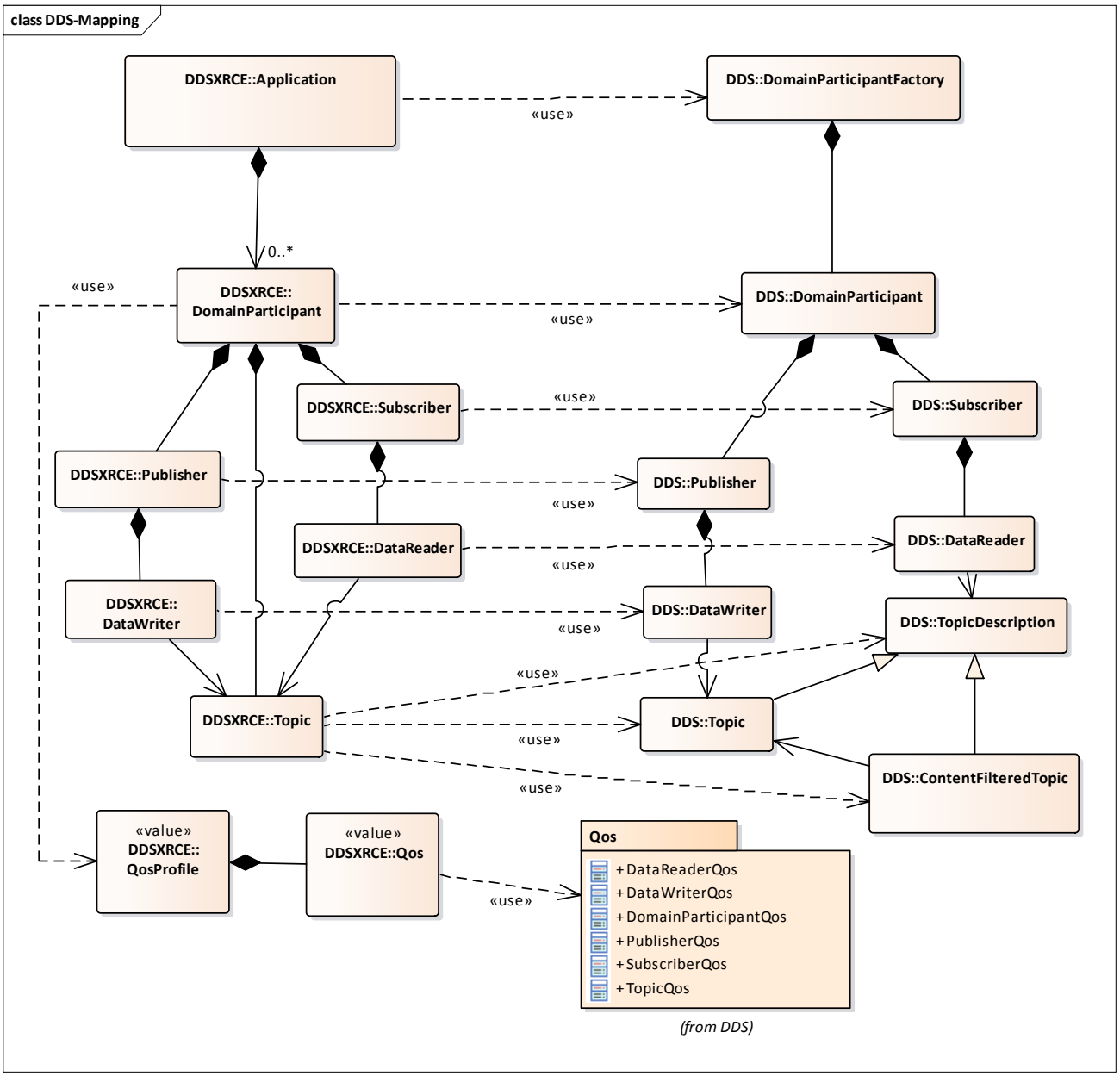


Figure 5 -- XRCE objects that proxy DDS Entities

7.6 XRCE Object Identification

Each XRCE Object managed by the XRCE Agent on behalf of a specific XRCE Client is identified by means of an `ObjectId`. This implies that the `ObjectId` shall be unique within the scope of an Agent and a `ClientKey`. The value of the `ObjectId` for a particular object shall be configured on the XRCE Agent or specified by the XRCE Client at the time the object is created.

There are two reserved values for `ObjectId`. The value {0x00, 0x00} is referred as OBJECTID_INVALID and represents an invalid object. The value {0xFF, 0xFF} is referred as OBJECTID_CLIENT and represents the XRCE ProxyClient object.

Alternatively, objects may also be identified by a string *resourceName*. The format of this name depends on the resource and provides a way to refer to a resource configured on the agent using a configuration file or similar means.

7.7 Data types used to model operations on XRCE Objects

The operations on the XRCE objects accept parameters. The format of these parameters is described as a set of IDL data types. These IDL descriptions are used in the description of the XRCE Object operations as well as used to define the wire representation of the messages exchanged between the Client and the Agent.

The IDL definitions for the data types shall be as specified in Annex A IDL Types. When serializing these types into a binary representation the encoding shall follow the rules defined in in [DDS-XTYPES] for XCDR version 2 encoding.

The following sub clauses provide explanations for some of the key data types specified in Annex A IDL Types.

7.7.1 Data and Samples

When the XRCE Agent sends data to the XRCE Client, it may use one of five possible formats. The formats differ depending on whether the data is sent by itself or accompanied by meta-data such as timestamp and sequence numbers. Another difference is whether the message contains a single sample or a sequence of samples.

While it would be possible to combine all of these representations into a single type (e.g. a union), doing so would introduce additional overhead in the serialization. This overhead is undesirable in bandwidth-constrained environments.

The five possible representation are: `SampleData`, `Sample`, `SampleDataSeq`, `SampleSeq`, and `SamplePackedData`. They respectively correspond to the `DataFormat` values FORMAT_DATA, FORMAT_DATA_SEQ, FORMAT_SAMPLE, FORMAT_SAMPLE_SEQ, and FORMAT_PACKED. Their IDL definition shall be as specified in Annex A IDL Types.

All of these representations serialize the data using the XCDR representation defined in [DDS-XTYPES]. For example, the definition of the `SampleData` is given by the IDL:

```
@extensibility(FINAL)
struct SampleData {
    XCDRSerializedBuffer serialized_data;
};
```

In this structure the `XCDRSerializedBuffer` represents the bytes resulting from serializing the application-specific data type that is being sent using the XCDR version 2 rules defined in clause 7.4 of [DDS-XTYPES].

Other representations include additional information but still rely on a `SampleData` to hold the serialized application-specific data. For example, the `DataFormat` FORMAT_SAMPLE uses the IDL type `Sample` defined below:

```
@bit_bound(8)
bitmask SampleInfoFlags {
    @position(0) INSTANCE_STATE_UNREGISTERED,
    @position(1) INSTANCE_STATE_DISPOSED,
    @position(2) VIEW_STATE_NEW,
    @position(3) SAMPLE_STATE_READ,
};
```

```
@extensibility(FINAL)
struct SampleInfo {
    SampleInfoFlags state;  //Combines SampleState, InstanceState, ViewState
    unsigned long   sequence_number;
    unsigned long   session_time_offset; // milliseconds up to 53 days
};

@extensibility(FINAL)
struct Sample {
    SampleInfo   info;
    SampleData   data;
};
```

The most compact `DataFormat` that includes sample information is FORMAT_PACKED. This format uses the IDL type `PackedSamples` defined below:

```
typedef unsigned short  DeciSecond; // 10e-1 seconds
@extensibility(FINAL)
struct SampleInfoDelta {
  SampleInfoFlags state;  // Combines SampleState, InstanceState, ViewState
  octet           seq_number_delta;
  DeciSecond      timestamp_delta; // In 1/10 of seconds
};
@extensibility(FINAL)
struct SampleDelta {
  SampleInfoDelta   info_delta;
  SampleData        data;
};
@extensibility(FINAL)
struct PackedSamples {
  SampleInfo             info_base;
  sequence<SampleDelta>  sample_delta_seq;
};
```

7.7.2 DataRepresentation

The `DataRepresentation` type is used to hold values of data samples as well as additional sample information, such as sequence number or timestamps. It is used by the `XRCE ProxyClient` **write** operation.

The `DataRepresentation` is defined as a union discriminated by a `DataFormat`. Depending on the discriminator it selects one of the formats defined in clause 7.7.1.

The possible values for the `DataFormat` and the resulting representation are described in Table 3 below.

Table 3 Interpretation of the DataFormat

DataFormat	*Selected DataRepresentation*
FORMAT_DATA	`struct SampleData` defined in Annex A IDL Types. Contains the data for a single sample without additional sample information.
FORMAT_DATA_SEQ	`struct SampleDataSeq` defined in Annex A IDL Types. Contains a sequence of data samples. Each data sample contains only the data without additional sample information.
FORMAT_SAMPLE	`struct Sample` defined in Annex A IDL Types. Contains a single sample with both the data and the additional sample information (`SampleInfo`). The `SampleInfo` holds the DDS `InstanceState`, `SampleState`, and `ViewState` of the corresponding DDS `Sample`. It also contains the sample sequence number and timestamp. The timestamp is represented as an offset relative to the session timestamp established when the session was created. The session timestamp corresponds to the ***client_timestamp*** attribute in `CLIENT_Representation`; see 7.8.2.1 and Annex A IDL Types.
FORMAT_SAMPLE_SEQ	`struct SampleSeq` defined in Annex A IDL Types. Contains a sequence of samples, each containing both the data and the additional sample information.
FORMAT_PACKED	`struct PackedSamples` defined in Annex A IDL Types. Contains a sequence of samples, each containing both the data and the additional sample information but using a more compact representation than `SampleSeq`. This representation is limited to samples that are close in sequence number (no more than 256 apart) and timestamp (100 minutes). It also uses timestamps with lower resolution (1/10 sec). The type `PackedSamples` contains a common `SampleInfo` (***info_base***) and a sequence of `SampleDelta`. Each `SampleDelta` contains a `SampleData` as well as an associated `SampleInfoDelta` (***info_delta***). The `SampleInfo` for each sample shall be computed by combining the common ***info_base*** with the ***info_delta*** that corresponds to that sample. The resulting SampleInfo (***resulting_info***) is defined as: ***resulting_info.state := info_delta.state*** ***resulting_info.sequence_number :=*** ***info_base.sequence_number + info_delta.seq_number_delta*** ***resulting_info. session_time_offset :=*** ***info_base. session_time_offset + info_delta.timestamp_delta***

The DataRepresentation type shall be as specified in Annex A IDL Types:

```
@extensibility(FINAL)
union DataRepresentation switch(DataFormat) {
  case FORMAT_DATA:
    SampleData data;
  case FORMAT_SAMPLE:
    Sample sample;
  case FORMAT_DATA_SEQ:
    SampleDataSeq data_seq;
  case FORMAT_SAMPLE_SEQ:
    SampleSeq sample_seq;
  case FORMAT_PACKED_SAMPLES:
    PackedSamples packed_samples;
};
```

7.7.3 ObjectVariant

The ObjectVariant type is used to hold the representation of a XRCE Object. It is used by the XRCE ProxyClient **create**, **update**, and **get_info** operations.

The ObjectVariant type is defined as a union discriminated by ObjectKind. Each value of the discriminator selects an appropriate object representation for that kind. See struct ObjectVariant defined in Annex A IDL Types.

For a given ObjectKind the, ObjectVariant type also supports multiple representation formats. Each format is identified by a value of the RepresentationFormat. Some formats are optimized for expressiveness and ease of configuration whereas others minimize the size used to transmit the representation.

The next sub clause defines the three possible formats; subsequent sub clauses provide details of the ObjectVariant representation for each kind of object and for each format.

7.7.3.1 Object Representation Formats

There are three RepresentationFormat values: REPRESENTATION_BY_REFERENCE, REPRESENTATION_AS_XML_STRING, and REPRESENTATION_IN_BINARY.

Some object kinds support all three formats; in this case the corresponding representation extends the type struct OBJK_Representation3_Base. Other object kinds support only two formats and therefore extend the type struct OBJK_RepresentationRefAndXML_Base or the type OBJK_RepresentationBinAndXML_Base.

These types are defined by the IDL below; see also Annex A, IDL Types.

```
const long REFERENCE_MAX_LEN  = 128;


@extensibility(FINAL)
union OBJK_Representation3Formats switch(RepresentationFormat) {
  case REPRESENTATION_BY_REFERENCE :
    string<REFERENCE_MAX_LEN>  object_reference
```

```
  case REPRESENTATION_AS_XML_STRING :
    string           xml_string_representation;
  case REPRESENTATION_IN_BINARY :
    sequence<octet>  binary_representation;
};

@extensibility(FINAL)
union OBJK_RepresentationRefAndXMLFormats switch(RepresentationFormat) {
  case REPRESENTATION_BY_REFERENCE :
    string<REFERENCE_MAX_LEN>  object_reference;
  case REPRESENTATION_AS_XML_STRING :
    string           string_representation;
};

@extensibility(FINAL)
union OBJK_RepresentationBinAndXMLFormats switch(RepresentationFormat) {
  case REPRESENTATION_IN_BINARY :
    sequence<octet>  binary_representation;
  case REPRESENTATION_AS_XML_STRING :
    string           string_representation;
};

@extensibility(FINAL)
struct OBJK_RepresentationRefAndXML_Base {
  OBJK_RepresentationRefAndXMLFormats representation;
};

@extensibility(FINAL)
struct OBJK_RepresentationBinAndXML_Base {
  OBJK_RepresentationBinAndXMLFormats representation;
};

@extensibility(FINAL)
struct OBJK_Representation3_Base {
  OBJK_Representation3Formats representation;
};
```

It is expected that additional representations may be added after they are defined in other OMG specifications. For example, there is ongoing work on a DDS-JSON RFP that would define a JSON format for describing DDS resources analogous to the XML format defined by the [DDS-XML] specification. This could be added as an additional REPRESENTATION_AS_JSON_STRING representation.

7.7.3.1.1 REPRESENTATION_BY_REFERENCE format

The `REPRESENTATION_BY_REFERENCE` represents the object using an ***object_reference*** encoded in a string. The string shall refer by name to a description already known to the XRCE `Agent`.

This format may be used to represent any object in an extremely compact manner. However it requires pre-configuration of the XRCE `Agent`. The pre-configuration may be done off-line prior to starting the XRCE `Agent` or may be done on-line using the DDS-XRCE protocol in combination with the `REPRESENTATION_AS_XML_STRING`.

The ***object_reference*** shall be a string formatted as defined by the XSD simpleType `elementNameReference` defined in the [DDS-XML] specification file **dds-xml_domain_definitions_nonamespace.xsd**.

It is expected that most XRCE `Clients` will use the ***object_reference*** to create resources in the XRCE `Agent`. This is because client applications are deployed as part of a system, and the system configuration and management process can configure the XRCE `Agent` for the intended deployment.

The following string is an example of an ***object_reference*** used to represent a XRCE `QosProfile`: `"MyQosLibrary::MyQosProfile"`.

This format is available for the XRCE `Object` kinds that can be configured as libraries in the XRCE `Agent`. These are XRCE `Type`, `QosProfile`, `Domain`, `DomainParticipan`, and `Application`.

7.7.3.1.2 REPRESENTATION_AS_XML_STRING format

The `REPRESENTATION_AS_XML_STRING` represents the object using an ***xml_string_representation*** string. The string shall contain an XML element formatted according to the [DDS-XML] specification. The format of the string is defined for each Object kind in clauses 7.7.3.2 to 7.7.3.11.

This format may be used to dynamically represent any XRCE `Object`. The disadvantage of this format is that it is more verbose due to the use of XML.

This format is intended for remotely configuring the agent. Typically it will not be used by the XRCE `Clients` except in deployments where the client-to-agent connection has sufficient bandwidth.

The following XML string is an example of a `REPRESENTATION_AS_XML_STRING` for the XRCE object `QosProfile`:

```
"<qos_library name='MyQosLibrary'>
  <qos_profile name='MyQosProfile'>
    <data_reader_qos>
      <reliability><kind>RELIABLE_RELIABILITY_QOS</kind><reliability>
      <time_based_filter>
        <minimum_separation><sec>10</sec></minimum_separation>
      </time_based_filter>
    <data_reader_qos>
  </qos_profile>
<qos_library>"
```

7.7.3.1.3 REPRESENTATION_IN_BINARY format

The `REPRESENTATION_IN_BINARY` represents objects using a ***binary_representation*** octet sequence. The octet sequence is the result of serializing an IDL-defined data-structure that depends on the kind of object using the XCDR version 2 format defined in [DDS-XTYPES].

This representation has the advantage of being very compact, but it can only be used to represent a subset of the XRCE `Objects`. Moreover not all DDS QoS can be expressed using the binary representation.

For example, the ***binary_representation*** for XRCE `Topic` is obtained by serializing an object of type `struct OBJK_Topic_Binary` defined in Annex A, IDL Types:

```
@extensibility(FINAL)
struct  OBJK_Topic_Binary {
    string<256> topic_name;
    @optional string<256> type_reference
    @optional DDS:XTypes::TypeIdentifier type_identifier;
};
```

For example, assuming little endian encoding, for a Topic with ***topic_name*** "Square" and ***type_reference*** "MyTypes::ShapeType" the ***binary_representation*** octet sequence would contain the 36 bytes:

```
{ 0x07, 0x00, 0x00, 0x00,
  'S',  'q',  'u',  'e',
  'r',  'e',  '\0', 0x01,
  0x13, 0x00, 0x00, 0x00,
  'M',  'y',  'T',  'y',
  'p',  'e',  's',  ':',
  ':',  'S',  'h',  'a',
  'p',  'e',  'T',  'y',
  'p',  'e',  '\0', 0x00 }
```

In the above note, the length of the two strings is 7 and 19 (in hexadecimal, 0x7 and 0x13), which are encoded in little endian so the least significant byte appears first.

Note also that the boolean value true (0x01) appears before the serialization of the ***type_reference*** indicating the presence of the optional member. The boolean value false (0x00) at the end indicates that the optional member ***type_identifier*** is not present.

7.7.3.2 XRCE QosProfile

The `OBJK_QOSPROFILE_Representation` supports the REPRESENTATION_BY_REFERENCE and REPRESENTATION_AS_XML_STRING formats. It is defined in Annex A, IDL Types as:

```
@extensibility(FINAL)
struct OBJK_QOSPROFILE_Representation :  OBJK_RepresentationRefAndXML_Base {
};
```

7.7.3.2.1 Representation by reference

When using the REPRESENTATION_BY_REFERENCE the ***object_reference*** field shall contain the fully qualified name of a `QosProfile` known to the XRCE Agent. The fully qualified name is composed of the name of the Qos library and the name of the QosProfile within the library. For example: `"MyLibrary::MyProfile"`.

7.7.3.2.2 XML string representation

When using the REPRESENTATION_AS_XML_STRING the ***string_representation*** field shall contain a single `<qos_library>` top-level XML element with the syntax defined by the XSD complexType `qosLibrary` defined in the [DDS-XML] machine-readable file **dds-xml_qos_definitions.xsd**. The `<qos_library>` element shall contain a single `<qos_profile>` child element.

The REPRESENTATION_AS_XML_STRING representation may reference other QoS profiles already known to the Agent. This feature also allows a compact way to represent a QosProfile that differs slightly from an existing one.

For example, the following XML defines a profile QosProfile called "MyQosLib::ModifiedProfile" that is based on an already defined profile "MyQosLib::MyQosProfile":

```
<qos_library name="MyQosLib">
    <qos_profile name="ModifiedProfile" base_name="MyQosLib:MyQosProfile">
        <data_reader_qos>
            <reliability><kind>RELIABLE_RELIABILITY_QOS</kind></reliability>
        </data_reader_qos>
    </qos_profile>
</qos_library>
```

The ***string_representation*** may reference other Qos Profiles already known to the XRCE Agent.

7.7.3.3 XRCE Type

The OBJK_TYPE_Representation supports the REPRESENTATION_BY_REFERENCE and REPRESENTATION_AS_XML_STRING formats. It is defined in Annex A, IDL Types as:

```
@extensibility(FINAL)
struct OBJK_TYPE_Representation        : OBJK_RepresentationRefAndXML_Base {
};
```

7.7.3.3.1 Representation by reference

When using the REPRESENTATION_BY_REFERENCE, the ***object_reference*** field shall contain the fully qualified name of a XRCE Type known to the XRCE Agent. The fully qualified name is composed of the name of the type prepended by the names of the enclosing modules. For example: "MyModule::ShapeType".

7.7.3.3.2 XML string representation

When using the REPRESENTATION_AS_XML_STRING, the ***string_representation*** field shall contain a single <types> top-level XML element representation with the syntax defined by the XSD complexType typeLibrary defined in the [DDS-XML] machine-readable file **dds-xml_type_definitions_nonamespace.xsd**.

Within the <types> element there may be multiple types defined. In this case only one type shall have the nested annotation (see [DDS-XTYPES]) set to false. This corresponds to the XRCE Type being created. Any types with nested annotation set to true, if present, may be used to represent the dependent types.

For example, the following XML defines a structure data-type "ShapeType" inside a module named "MyModule" referenceable as "MyModule::ShapeType":

```
<types>
    <module name="MyModule">
      <struct name="ShapeType">
        <member name="color" key="true" type="string" stringMaxLength="32"/>
        <member name="x" type="int32" />
        <member name="y" type="int32" />
        <member name="shapesize" type="int32" />
      </struct>
    </module>
</types>
```

The ***string_representation*** may reference other Types already known to the Agent.

7.7.3.4 XRCE Domain

The OBJK_DOMAIN_Representation supports the REPRESENTATION_BY_REFERENCE and REPRESENTATION_AS_XML_STRING formats. It is defined in Annex A, IDL Types as:

```
@extensibility(FINAL)
struct OBJK_DOMAIN_Representation : OBJK_RepresentationRefAndXML_Base {
};
```

7.7.3.4.1 Representation by reference

When using the REPRESENTATION_BY_REFERENCE, the ***object_reference*** field shall contain the fully qualified name of a XRCE `Domain` definition known to the Agent. The fully qualified name is composed of the name of the Domain library and the name of the Domain within the library. For example: "`MyDomainLib::ShapesDomain`".

7.7.3.4.2 XML string representation

When using the REPRESENTATION_AS_XML_STRING, the ***string_representation*** field shall contain the XML representation of a `Domain` as defined in [DDS-XML]. The XML shall contain a single `<domain_library>` top-level XML element with the syntax defined by the XSD complexType `qosDomain` defined in the [DDS-XML] machine-readable file **dds-xml_domain_definitions_nonamespace.xsd**. The `<domain_library>` element shall contain a single `<domain>` child element.

For example, the following XML defines a domain referenceable as "`MyDomainLib::ShapesDomain`".

```
<domain_library name="MyDomainLib">
    <domain name="ShapesDomain" domain_id="0">
      <register_type name="ShapeType" type_ref="ShapeType" />
      <topic name="Square" register_type_ref="ShapeType" />
    </domain>
</domain_library>
```

The ***string_representation*** may reference Types already known to the XRCE Agent.

7.7.3.5 XRCE Application

The `OBJK_TYPE_Representation` supports the REPRESENTATION_BY_REFERENCE and REPRESENTATION_AS_XML_STRING formats. It is defined in Annex A, IDL Types as:

```
@extensibility(FINAL)
struct OBJK_APPLICATION_Representation : OBJK_RepresentationRefAndXML_Base {
};
```

7.7.3.5.1 Representation by reference

When using the REPRESENTATION_BY_REFERENCE, the ***object_reference*** field shall contain the fully qualified name of a XRCE Application definition known to the Agent. The fully qualified name is composed of the name of the Application library and the name of the Application within the library. For example: "`MyAppLibrary::ShapePublisherApp`".

7.7.3.5.2 XML string representation

When using the REPRESENTATION_AS_XML_STRING, the ***string_representation*** field shall contain the XML representation of an `Application` as defined in [DDS-XML]. The XML shall contain a single `<application_library>` top-level XML element with the syntax defined by the XSD complexType `applicationLibrary` defined in the [DDS-XML] machine-readable file **dds-xml_application_definitions_nonamespace.xsd**. This element shall contain a single `<application>` child element.

For example, the following XML defines an application referencable as “MyAppLibrary::ShapePublisherApp”:

```
<application_library name="MyAppLibrary">
    <application name="ShapePublisherApp">
        <domain_participant name="MyParticipant1" domain_id="0">
            <register_type name="ShapeType" type_ref="MyTypes::ShapeType" />
            <topic register_type_ref="ShapeType" name="Square" />
            <publisher name="MyPublisher">
              <data_writer name="MyWriter"  topic_ref="Square" />
            </publisher>
        </domain_participant>
        <domain_participant name="MyParticipant2" domain_id="0">
            <register_type name="ShapeType" type_ref="MyTypes::ShapeType" />
            <topic register_type_ref="ShapeType" name="Square" />
            <subscriber name="MySubscriber">
                <data_writer name="MyReader"  topic_ref="Circle" />
            </subscriber>
        </domain_participant>
    </application>
</application_library>
```

The ***string_representation*** may reference XRCE Types, Qos Profiles, Domains, or DomainParticipants already known to the XRCE Agent.

7.7.3.6 XRCE DomainParticipant

The `OBJK_PARTICIPANT_Representation` supports three representation formats. It is defined in Annex A, IDL Types as:

```
@extensibility(FINAL)
struct OBJK_PARTICIPANT_Representation : OBJK_Representation3_Base {
};
```

7.7.3.6.1 Representation by reference

When using the REPRESENTATION_BY_REFERENCE, the ***object_reference*** field shall contain the fully qualified name of a XRCE `DomainParticipant` definition known to the Agent. The fully qualified name is composed of the name of the DomainParticipant library and the name of the DomainParticipant within the library. For example: "`MyParticipantLibrary::ShapePublisherApp`".

7.7.3.6.2 XML string representation

When using the REPRESENTATION_AS_XML_STRING, the ***string_representation*** field shall contain a single `<domain_participant_library>` top-level XML element with the syntax defined by the XSD complexType `domainParticipantLibrary` defined in the [DDS-XML] machine-readable file **dds-xml_domain_participant_definitions_nonamespace.xsd**. This element shall contain a single `<domain_participant>` child element.

For example, the following XML string defines a DDS-XML `DomainParticipant` referenceable as "`MyParticipantLibrary::MyParticipant`".

```
<domain_participant_library name="MyParticipantLibrary">
    <domain_participant name="MyParticipant" domain_id="0">
        <register_type name="ShapeType" type_ref="MyTypes::ShapeType" />
        <topic register_type_ref="ShapeType" name="Square" />
        <publisher name="MyPublisher">
            <data_writer name="MyWriter"  topic_ref="Square" />
        </publisher>
    </domain_participant>
</domain_participant_library>
```

The ***string_representation*** may reference XRCE Types, Qos Profiles, Domains, or DomainParticipants already known to the XRCE Agent.

7.7.3.6.3 Binary representation

When using the REPRESENTATION_IN_BINARY, the ***binary_representation*** octet sequence shall contain the XCDR version 2 serialized representation [DDS-XTYPES] of the structure `OBJK_DomainParticipant_Binary` defined in Annex A IDL Types.

```
@extensibility(FINAL)
struct  OBJK_DomainParticipant_Binary {
    long domain_id;
    @optional string<128> domain_reference;
    @optional string<128> qos_profile_reference;
};
```

The optional ***domain_reference*** field may be used to reference a XRCE `Domain` definition known to the Agent. It shall the representation by reference of the domain as defined in 7.7.3.4.1. For example: `"MyDomainLib::ShapesDomain"`.

Any XRCE Topic and Type definitions contained in the referenced domain are considered defined within the scope of the XRCE `DomainParticipant` and become available as references to construct XRCE objects contained by the `DomainParticipant`.

The optional ***qos_profile_reference*** field may be used to reference a XRCE `QosProfile` definition known to the Agent. It shall contain the representation by reference of the QosProfile defined in 7.7.3.2.1. For example: "`MyQosLib:MyQosProfile`". If specified, the corresponding DDS `DomainPaticipant` shall be created using that Qos. Otherwise, the `DomainPaticipant` shall be created using the DDS default Qos.

7.7.3.7 XRCE Topic

The `OBJK_TOPIC_Representation` supports three representation formats. It is defined in Annex A, IDL Types as:

```
@extensibility(FINAL)
struct OBJK_TOPIC_Representation : OBJK_Representation3_Base {
    ObjectId participant_id;
};
```

Independent of the representation format, the field ***participant_id*** shall contain the `ObjectId` of a XRCE `DomainParticipant` object. The referenced or created `Topic` will belong to the specified `DomainParticipant`.

7.7.3.7.1 Representation by reference

When using the REPRESENTATION_BY_REFERENCE, the ***object_reference*** field shall contain the bare name of a XRCE `Topic` defined in XRCE `DomainParticipant` identified by the ***participant_id***. The `Topic` could be defined directly on the XRCE `DomainParticipant`, or else in the XRCE `Domain` associated with the `DomainParticipant`.

For example, if the `DomainParticipant` had been defined with a reference to the XRCE `Domain` "`MyDomainLib::ShapesDomain`" shown as an example in 7.7.3.4.2, then the ***object_reference*** "`Square`" could be used to reference the namesake Topic of type "`ShapeType`" defined there.

7.7.3.7.2 XML string representation

When using the REPRESENTATION_AS_XML_STRING, the ***string_representation*** field shall contain a single `<topic>` top-level XML element with the syntax defined by the XSD complexType `topic` defined in the [DDS-XML] machine-readable file **dds-xml_domain_definitions_nonamespace.xsd**.

For example, the following XML string defines a DDS-XML `Topic` with name "`Square`".

```
<topic name="Square" register_type_ref="ShapeType" />
```

The ***string_representation*** may reference XRCE Types or QosProfiles already known to the XRCE Agent.

7.7.3.7.3 Binary representation

When using the REPRESENTATION_IN_BINARY, the ***binary_representation*** octet sequence shall contain the XCDR version 2 serialized representation [DDS-XTYPES] of the structure `OBJK_Topic_Binary` defined in Annex A IDL Types:

```
@extensibility(FINAL)
struct  OBJK_Topic_Binary {
    string<256> topic_name;
    @optional string<256> type_reference;
    @optional DDS:XTypes::TypeIdentifier type_identifier;
};
```

Either ***type_reference*** or ***type_identifier*** may be used to identify the XRCE Type associated with the Topic. Either member may be omitted, but not both. If both are present the ***type_identifier*** shall take precedence.

The ***type_identifier***, if present, shall contain the DDS-XTYPES `TypeIdentifier` for the data-type. See clause 7.3.2 of [DDS-XTYPES].

The ***type_reference***, if present, shall contain the fully qualified name of the type, including containing modules as specified in 7.7.3.3.1. The referenced type shall be known to the XRCE Agent either via pre-configuration, or as a result of a prior **create** operation executed on the XRCE `ProxyClient`; see 7.8.3.1.

7.7.3.8 XRCE Publisher

The `OBJK_PUBLISHER_Representation` supports the REPRESENTATION_IN_BINARY and REPRESENTATION_AS_XML_STRING formats. It is defined in Annex A, IDL Types as:

```
@extensibility(FINAL)
struct OBJK_PUBLISHER_Representation   : OBJK_RepresentationBinAndXML_Base {
    ObjectId participant_id;
};
```

Independent of the representation format, the member ***participant_id*** shall contain the `ObjectId` of a XRCE `DomainParticipant` object. The referenced or created `Publisher` shall belong to the specified `DomainParticipant`.

7.7.3.8.1 XML string representation

When using the REPRESENTATION_AS_XML_STRING, the ***string_representation*** field shall contain a single `<publisher>` top-level XML element with the syntax defined by the XSD complexType `publisher` defined in the [DDS-XML] machine-readable file **dds-xml_domain_participant_definitions_nonamespace.xsd**.

For example, the following XML string defines a XML `Publisher` referenceable within the XRCE `DomainParticipant` as "`MyPublisher`".

```
<publisher name="MyPublisher"/>
```

Note that the XML representation of a `Publisher` allows specifying Qos policies and including nested `DataWriter` objects. These additional definitions may reference other XRCE objects (Qos profiles or topics). Any referenced object must have been previously created or configured on the XRCE `Agent`. For example, the following XML string defines a XRCE `Publisher` with a Qos and a contained `DataWriter`:

```
<publisher name="MyPublisher"/>
      <publisher_qos base_name="MyQosLib:MyProfile" />
      <data_writer name="MySquareWriter" topic_ref="Square" />
</publisher>
```

7.7.3.8.2 Binary representation

When using the REPRESENTATION_IN_BINARY, the ***binary_representation*** shall contain the XCDR version 2 serialized representation [DDS-XTYPES] of the structure `OBJK_Publisher_Binary` defined in A IDL Types:

```
@extensibility(FINAL)
struct  OBJK_PUBLISHER_QosBinary {
    @optional sequence<string>  partitions;
    @optional sequence<octet>   group_data;
};
@extensibility(FINAL)
struct  OBJK_Publisher_Binary {
    @optional string publisher_name;
    @optional OBJK_PUBLISHER_QosBinary  qos;
};
```

7.7.3.9 XRCE Subscriber

The `OBJK_SUBSCRIBER_Representation` supports the REPRESENTATION_IN_BINARY and REPRESENTATION_AS_XML_STRING formats. It is defined in Annex A, IDL Types as:

```
@extensibility(FINAL)
struct OBJK_SUBSCRIBER_Representation   : OBJK_RepresentationBinAndXML_Base {
    ObjectId participant_id;
};
```

Independent of the representation format, the member ***participant_id*** shall contain the `ObjectId` of a XRCE `DomainParticipant` object. The referenced or created `Subscriber` shall belong to the specified `DomainParticipant`.

7.7.3.9.1 XML string representation

When using the REPRESENTATION_AS_XML_STRING, the ***string_representation*** field shall contain a single `<subscriber>` top-level XML element with the syntax defined by the XSD complexType `subscriber` defined in the [DDS-XML] machine-readable file **dds-xml_domain_participant_definitions_nonamespace.xsd**.

For example, the following XML string defines a XRCE Subscriber referenceable within the `DomainParticipant` as "`MySubscriber`":

```
<subscriber name="MySubscriber"/>
```

Note that the XML representation of a Subscriber allows specifying Qos policies and including nested DataReader objects. These additional definitions may reference other XRCE objects (Qos profiles or topics). Any referenced object must have been previously created or configured on the XRCE Agent. For example, the following XML string defines a XRCE Subscriber with a Qos and a contained DataReader:

```
<subscriber name="MySubscriber"/>
    <subscriber_qos base_name=”MyQosLib:MyProfile” />
    <data_reader name="MySquareReader" topic_ref="Square" />
</subscriber>
```

7.7.3.9.2 Binary representation

When using the REPRESENTATION_IN_BINARY, the ***binary_representation*** shall contain the XCDR version 2 serialized representation [DDS-XTYPES] of the structure OBJK_Subscriber_Binary defined in Annex A IDL Types.

```
@extensibility(FINAL)
    struct  OBJK_SUBSCRIBER_QosBinary {
    @optional sequence<string>  partitions;
    @optional sequence<octet>   group_data;
};
@extensibility(FINAL)
struct  OBJK_Subscriber_Binary {
    @optional string subscriber_name;
    @optional OBJK_SUBSCRIBER_QosBinary  qos;
};
```

7.7.3.10 XRCE DataWriter

The DATAWRITER_Representation supports the REPRESENTATION_IN_BINARY and REPRESENTATION_AS_XML_STRING formats. It is defined in Annex A, IDL Types as:

```
@extensibility(FINAL)
struct DATAWRITER_Representation    : OBJK_RepresentationBinAndXML_Base {
    ObjectId publisher_id;
};
```

Independent of the representation format, the member ***publisher_id*** shall contain the ObjectId of a XRCE Publisher object. The referenced or created DataWriter shall belong to the specified Publisher.

7.7.3.10.1 XML string representation

When using the REPRESENTATION_AS_XML_STRING, the ***string_representation*** field shall contain a single <data_writer> top-level XML element with the syntax defined by the XSD complexType dataWriter defined in the [DDS-XML] machine-readable file **dds-xml_domain_participant_definitions_nonamespace.xsd**.

For example, the following XML string defines a XRCE DataWriter for Topic "Square" referenceable within the XRCE Susbcriber as "MySquareWriter":

```
<data_writer name="MySquareWriter" topic_ref="Square"/>
```

The referenced Topic must have been previously created or configured on the XRCE DomainParticipant to which the Publisher and DataWriter belong.

The XML representation of a DataWriter allows specifying Qos policies. These may reference other XRCE (Qos profiles. Any referenced object must have been previously created or configured on the XRCE Agent. For example, the following XML string defines a XRCE DataWriter with a Qos that extends the profile "MyQosLib:MyProfile" additionally setting the DEADLINE Qos policy.

```
<data_writer name="MySquareWriter" topic_ref="Square">
    <data_writer_qos base_name="MyQosLib::MyProfile">
        <deadline>
            <period><sec>120</sec></period>
        </deadline>
    </data_writer_qos>
</data_writer>
```

7.7.3.10.2 Binary representation

When using the REPRESENTATION_IN_BINARY, the ***binary_representation*** shall contain the XCDR version 2 serialized representation [DDS-XTYPES] of the structure OBJK_DataWriter_Binary defined in Annex A IDL Types:

```
    @bit_bound(16)
    bitmask EndpointQosFlags {
        @position(0) is_reliable,
        @position(1) is_history_keep_last,
        @position(2) is_ownership_exclusive,
        @position(3) is_durability_transient_local,
        @position(4) is_durability_transient,
        @position(5) is_durability_persistent,
    };
    @extensibility(FINAL)
    struct  OBJK_Endpoint_QosBinary {
      EndpointQosFlags            qos_flags;
      @optional unsigned short    history_depth;
      @optional unsigned long     deadline_msec;
      @optional unsigned long     lifespan_msec;
      @optional sequence<octet>   user_data;
    };
    @extensibility(FINAL)
    struct  OBJK_DataWriter_Binary {
      string                      topic_name;
      OBJK_Endpoint_QosBinary     endpoint_qos;
      @optional unsigned long     ownership_strength;
  };
```

7.7.3.11 XRCE DataReader

The DATAREADER_Representation supports the REPRESENTATION_IN_BINARY and REPRESENTATION_AS_XML_STRING formats. It is defined in Annex A, IDL Types as:

```
@extensibility(FINAL)
struct DATAREADER_Representation     : OBJK_RepresentationBinAndXML_Base {
    ObjectId subscriber_id;
};
```

Independent of the representation format, the member ***subscriber_id*** shall contain the ObjectId of a XRCE Subscriber object. The referenced or created DataReader will belong to the specified Subscriber.

7.7.3.11.1 XML string representation

When using the REPRESENTATION_AS_XML_STRING, the ***string_representation*** field shall contain a single <data_reader> top-level XML element with the syntax defined by the XSD complexType dataReader defined in the [DDS-XML] machine-readable file **dds-xml_domain_participant_definitions_nonamespace.xsd**.

For example, the following XML string defines a XRCE DataReader for Topic "Square" referenceable within the XRCE Publisher as "MySquareReader":

```
<data_reader name="MySquareReader" topic_ref="Square"/>
```

The referenced Topic must have been previously created or configured on the XRCE DomainParticipant to which the Subscriber and DataReader belong.

The XML representation of a DataReader allows specifying Qos policies. These may reference other XRCE Qos profiles. Any referenced objects must have been previously created or configured on the XRCE Agent.

The XML representation of a DataReader may also contain time-based and content-based filters.

For example, the following XML string defines a XRCE DataReader with a Qos that extends the profile "MyQosLib:MyProfile" assing/setting the DEADLINE Qos policy and sets a content filter.

```
<data_reader name="MySquareReader" topic_ref="Square">
    <data_reader_qos base_name="MyQosLib:MyProfile">
        <deadline>
            <period><sec>120</sec></period>
        </deadline>
    </data_reader_qos>
    <content_filter name="MyFilter">
        <expression> x &gt; 5 </expression>
    </content_filter>
</data_reader>
```

7.7.3.11.2 Binary representation

When using the REPRESENTATION_IN_BINARY, the ***binary_representation*** shall contain the XCDR version 2 serialized representation [DDS-XTYPES] of the structure OBJK_DataReader_Binary defined in A IDL Types. See also Binary representation of the DataWriter in 7.7.3.10.2 for the definition of OBJK_Endpoint_QosBinary.

```
@extensibility(FINAL)
struct  OBJK_DataReader_Binary {
    string                      topic_name;
    OBJK_Endpoint_QosBinary     endpoint_qos;
    @optional unsigned long     timebasedfilter_msec;
```

```
    @optional string            contentbased_filter;
};
```

7.7.4 ObjectId

The XRCE `ObjectId` is used to hold the unique identification of an XRCE Object. Each `ObjectId` is scoped to an XRCE `Client` and `Agent` pair. Consequently, the `ObjectId` values managed by an `Agent` need to be unique only for each XRCE `Client`. An XRCE `Client` normally connects to a single XRCE `Agent`. In this situation, the XRCE `Client` can treat the `ObjectId` as globally unique.

The `ObjectId` is defined in A IDL Types as:

```
typedef octet ObjectId     [2];
```

7.7.5 ObjectKind

The XRCE `ObjectKind` is used to enumerate and identify the kind of XRCE Object. XRCE objects are classified into 14 kinds. The possible kinds are defined in A IDL Types as:

```
    typedef octet ObjectKind;

    const ObjectKind OBJK_INVALID     = 0x00;
    const ObjectKind OBJK_PARTICIPANT = 0x01;
    const ObjectKind OBJK_TOPIC       = 0x02;
    const ObjectKind OBJK_PUBLISHER   = 0x03;
    const ObjectKind OBJK_SUBSCRIBER  = 0x04;
    const ObjectKind OBJK_DATAWRITER  = 0x05;
    const ObjectKind OBJK_DATAREADER  = 0x06;
    const ObjectKind OBJK_TYPE        = 0x0A;
    const ObjectKind OBJK_QOSPROFILE  = 0x0B;
    const ObjectKind OBJK_APPLICATION = 0x0C;
    const ObjectKind OBJK_AGENT       = 0x0D;
    const ObjectKind OBJK_CLIENT      = 0x0E;
```

7.7.6 ObjectIdPrefix

The `ObjectIdPrefix` is used to hold the unique identification of an XRCE object of a specific `ObjectKind`. The `ObjectId` of an object is composed combining 12 bits from the `ObjectIdPrefix` and four bits from the `ObjectKind`.

The `ObjectIdPrefix` is defined in A IDL Types as:

```
typedef octet ObjectIdPrefix [2];
```

Assuming an XRCE object has ObjectIdPrefix ***objectid_prefix***, `ObjectKind` ***object_kind***, and `ObjectId` ***object_id*** the following relationships shall hold:

object_id[0] = objectid_prefix[0]

object_id[1] = (objectid_prefix[1]&0xF0) + object_kind

7.7.7 ResultStaus

The `ResultStatus` is used to hold the return value of the operations on the XCRE objects. It contains a `StatusValue` that encodes whether the operation succeeded or failed as well as the reason for the failure. It also contains a specialized implementation-specific status, which is used to return vendor or implementation-specific information.

The `StatusValue` and `ResultStatus` are defined in defined in Annex A IDL Types as:

```
@bit_bound(8)
enum StatusValue {
    @value(0x00) STATUS_OK,
    @value(0x01) STATUS_OK_MATCHED,
    @value(0x80) STATUS_ERR_DDS_ERROR,
    @value(0x81) STATUS_ERR_MISMATCH,
    @value(0x82) STATUS_ERR_ALREADY_EXISTS,
    @value(0x83) STATUS_ERR_DENIED,
    @value(0x84) STATUS_ERR_UNKNOWN_REFERENCE,
    @value(0x85) STATUS_ERR_INVALID_DATA,
    @value(0x86) STATUS_ERR_INCOMPATIBLE,
    @value(0x87) STATUS_ERR_RESOURCES
};
struct ResultStatus {
  StatusValue  status;
  octet        implementation_status;
};
```

The interpretation of the `StatusValue` is specified in below.

Table 4—Interpretation of StatusValue

StatusValue	*Interpretation*
STATUS_OK	Indicates a successful execution of the operation
STATUS_OK_MATCHED	Indicates a successful execution of a create or update operation on a resource when the resource already existed on the Agent and the resource state already matched the one requested by the operation. As a consequence, no actual change was made to the resource.
STATUS_ERR_DDS_ERROR	Indicates a failure in the execution of the operation caused by an error when creating or operating on the DDS resource related to the operation.
STATUS_ERR_MISMATCH	Indicates a failure in the execution of a create or update operation on a resource when the resource already existed on the Agent, the state did not match the one requested by the operation, and it was not possible to change the state of the resource.
STATUS_ERR_ALREADY_EXISTS	Indicates a failure in the execution of a create operation due to the fact that the resource already existed.
STATUS_ERR_DENIED	Indicates a failure in the execution of an operation due to lack of permissions.
STATUS_ERR_UNKNOWN_REFERENCE	Indicates a failure in the execution of an operation due to the fact that the referenced resource is not known to the Agent.
STATUS_ERR_INVALID_DATA	Indicates a failure in the execution due to wrong on invalid input parameter data.
STATUS_ERR_INCOMPATIBLE	Indicates a failure in the execution of an operation due to an incompatibility between the Client and the Agent.
STATUS_ERR_RESOURCES	Indicates a failure in the execution of an operation due to a resource error on the Agent.

7.7.8 BaseObjectRequest

The `BaseObjectRequest` type is used to hold the common parameters of the requests sent from the XRCE `Client` to the `Agent`. It is defined in Annex A IDL Types as:

```
@extensibility(FINAL)
struct BaseObjectRequest {
    RequestId     request_id;
    ObjectId      object_id;
};
```

The interpretation of the members of this type (i.e. parameters sent as part of the requests) shall be:

- ***request_id*** (RequestId) identifies each request. It is used to correlate a reply with the related request. It is scoped to each XRCE `Client` and `Agent` pair. Note that it is possible to reuse a value of the ***request_id*** for future

requests as long as the previous request with that value is known by `Client` and `Agent` to no longer be active.

- ***object_id*** (ObjectId) the `ObjectId` that is the target of the request. For requests that create objects, the ***object_id*** conveys the `ObjectIdPrefix` for the created object. See 7.7.6.

7.7.9 BaseObjectReply

The `BaseObjectReply` type is used to hold the common parameters of the replies sent from the XRCE `Agent` back to the `Client`. It is defined in defined in Annex A IDL Types as:

```
struct ResultStatus {
    StatusValue  status;
    octet        implementation_status;
};

@extensibility(FINAL)
struct BaseObjectReply {
    BaseObjectRequest  related_request;
    ResultStatus       result;
};
```

The interpretation of the members of these types (i.e. parameters sent as part of the requests) shall be:

- ***related_request*** contains the ***request_id*** and ***object_id*** of the request that caused the reply to be sent:
    - The ***request_id*** (RequestId) identifies the request. It is used to correlate a reply with the request.
    - The ***object_id*** (ObjectId) is the target of the request. For requests that create objects, the ***object_id*** conveys the desired `ObjectId` for the created object. In this case the ***object_id*** is interpreted as a prefix to be combined with the `ObjectKind` to obtain the final `ObjectId`.
- ***status*** (ResultStatus). Enumerated value indicating whether the related request operation succeeded or failed. If the operation succeeded the `StatusValue` shall be set to STATUS_OK or STATUS_OK_MATCHED. If it failed it shall be set to the value that corresponds to the type of error encountered.
- ***implementation_status*** (octet) provides an implementation-specific (vendor-specific) return status. The value is scoped by the `XrceVendorId` of the `Agent`. It shall only be interpreted by clients that understand the implementation status values of the `XrceVendorId` of the Agent that returned it.

7.7.10 RelatedObjectRequest

The `RelatedObjectRequest` type is used to hold the common parameters of the messages sent from the XRCE `Agent` back to the `Client` that are indirectly related to a prior request from the `Client`. For example, DATA messages that related to a previous **read** operation, see 7.8.5.1.

It is defined in Annex A IDL Types as:

```
typedef RelatedObjectRequest BaseObjectRequest;
```

The interpretation is the same as for the ***related_request*** that appears in the `BaseObjectReply`, see 7.7.9.

7.7.11 CreationMode

The `CreationMode` type is used to control the behavior of the `ProxyClient` **create** operation. See clause 7.8.3.1. It is defined in Annex A IDL Types as:

```
struct CreationMode {
    boolean reuse;
    boolean replace;
};
```

7.7.12 ActivityInfoVariant

The `ActivityInfoVariant` type is used to hold information on the activity of a XRCE object. It is used by the `ProxyClient` **get_info** operation. See clause 7.8.3.3. It is defined in Annex A IDL Types as:

```
bitmask InfoMask {
    @position(0) INFO_CONFIGURATION,
    @position(1) INFO_ACTIVITY
};

@extensibility(APPENDABLE)
struct AGENT_ActivityInfo {
    short availability;
    TransportLocatorSeq address_seq;
};

@extensibility(APPENDABLE)
struct DATAREADER_ActivityInfo {
    short highest_acked_num;
};

@extensibility(APPENDABLE)
struct DATAWRITER_ActivityInfo {
    unsigned long long sample_seq_num;
    short stream_seq_num;

};

@extensibility(FINAL)
union ActivityInfoVariant (ObjectKind) {
  case OBJK_DATAWRITER :
```

```
    DATAWRITER_ActivityInfo data_writer;
  case OBJK_DATAREADER :
    DATAREADER_ActivityInfo data_reader;
};
```

7.7.13 ObjectInfo

The `ObjectInfo` type is used to hold information on the configuration and activity of a XRCE object. It I used by the `ProxyClient` **get_info** operation. See clause 7.8.3.3. It is defined in Annex A IDL Types. See also clause 7.7.3 for a description of ObjectVariant and 7.7.12 for a description of ActivityInfoVariant.

```
@extensibility(FINAL)
struct ObjectInfo {
    @optional ActivityInfoVariant  activity;
    @optional ObjectVariant        config;

};
```

7.7.14 ReadSpecification

The `ReadSpecification` type is used to control the information returned by the `ProxyClient` **read** operation. See clause 7.8.5.1. It is defined in Annex A IDL Types as:

```
    @extensibility(APPENDABLE)
    struct DataDeliveryControl {
        unsigned short max_samples;
        unsigned short max_elapsed_time;
        unsigned short max_bytes_per_second;
        unsigned short min_pace_period;  // milliseconds
    };

    @extensibility(FINAL)
    struct ReadSpecification  {
        DataFormat data_format;
        @optional string              content_filter_expression;
        @optional DataDeliveryControl delivery_control;
    };
```

7.8 XRCE Object operations

7.8.1 Use of the ClientKey

All operations are performed within the context of a `ClientKey`, which is used both to authenticate and identify the client:

- The `ClientKey` is assigned to each client. The `ClientKey` uniquely identifies the client to a particular agent. The `ClientKey` is associated with a set of permissions for the client within the agent.
- The `ClientKey` shall be considered secret. It must be configured both on the `Client` and in the `Agent`. The creation and configuration are outside the scope of this specification.
- The `ClientKey` shall not be interpreted.

With the exception of the operations **create_client** and **get_info** on the XRCE Root, all other operations expect that the `ClientKey` references an already exiting XRCE `ProxyClient`. If this is not the case, the operation shall fail.

To avoid information leakage that could compromise security, the failure to locate a `ClientKey` may in some cases result in a ***returnValue*** having STATUS_ERR_NOCLIENT while in others it may silently drop the connection to the client.

The `Agent` shall maintain a counter on the number of times the STATUS_ERR_NOCLIENT was sent on an established connection, and once a certain threshold is crossed it shall close the connection. The `Agent` may subsequently refuse or throttle new connections originating from the same client transport endpoint that was previously closed. The specific details of this behavior are implementation-specific and left outside the scope of this specification.

7.8.2 XRCE Root

The XRCE `Root` object represents the `Agent`. An XRCE `Agent` is a singleton object that all agents shall instantiate.

The XRCE `Root` is responsible for authenticating client applications and creating the XRCE `ProxyClient` object associated with each client.

The logical operations on the XRCE Root are shown in Table 5.

Table 5-- XRCE Root operations

create_client		ResultStatus
	object_representation	CLIENT_Representation
	out: agent_info	AGENT_Representation
get_info		ResultStatus
	info_mask	InfoMask
	client_info	ObjectInfo
	out: agent_info	ObjectInfo
delete_client		ResultStatus

7.8.2.1 create_client

Inputs

- ***client_representation*** (CLIENT_Representation): a representation of the `Client`.

Outputs

- ***returnValue*** (ResultStatus): indicates whether the operation succeeded and the current status of the XRCE `ProxyClient` object.
- ***agent_info*** (AGENT_Representation): a representation of the `Agent`.

The ***client_representation*** shall contain a `CLIENT_Representation` which is used to initialize the XRCE `ProxyClient`. This type is defined in Annex A, IDL Types as:

```
@extensibility(FINAL)
struct CLIENT_Representation {
    XrceCookie   xrce_cookie;  // XRCE_COOKIE
    XrceVersion  xrce_version;
    XrceVendorId xrce_vendor_id;
    Time_t       client_timestamp;
    ClientKey    client_key;
    SessionId    session_id;
    @optional  PropertySeq properties;
};
```

The ***agent_representation*** shall contain an `AGENT_Representation` which informs the `Client` about the configuration of the `Agent`. This type is defined in Annex A, IDL Types as:

```
@extensibility(FINAL)
struct AGENT_Representation {
    XrceCookie   xrce_cookie;  // XRCE_COOKIE
    XrceVersion  xrce_version;
    XrceVendorId xrce_vendor_id;
    Time_t       agent_timestamp;
    @optional  PropertySeq  properties;
};
```

The `XRCE Agent` shall perform the following checks and actions based on the information found within the ***client_representation***:

- Check the ***xrce_cookie*** to ensure it matches the predefined XRCE_COOKIE constant. If it does not match the creation shall fail and set the returnValue StatusValue to STATUS_ERR_INVALID_DATA.
- Check that the major version (***xrce_version[0]***) matches the XRCE_VERSION_MAJOR. If it does not match, the creation shall fail and set the ***returnValue*** StatusValue to STATUS_ERR_INCOMPATIBLE.
- Check that the `Client` identified by the ***client_key*** is authorized to connect to the `XRCE Agent`. If this check fails the operation shall fail and set the ***returnValue*** StatusValue to STATUS_ERR_DENIED.
- Check the `Client` ***properties***, if present. These may contain vendor-specific information that may prevent the Agent from accepting the connection from the `Client`. The ***properties*** field may include extra authentication tokens (e.g. username and password) or other configuration information. If this check fails the operation shall fail and set the ***returnValue*** StatusValue to the appropriate value.
- Check if there is an existing `XRCE ProxyClient` object associated with the same ***client_key*** and, if so, compare the ***session_id*** of the existing `ProxyClient` with the one in the ***client_representation***:
    - If a `ProxyClient` exists and has the same ***session_id***, then the operation shall not perform any action and shall set the ***returnValue*** StatusValue to STATUS_OK.
    - If a `ProxyClient` exists and has a different ***session_id*** then the operation shall delete the existing `XRCE ProxyClient` object and subsequently take the same actions as if there had not been a `ProxyClient` associated with the ***client_key***.
- Check that there are sufficient internal resources to complete the create operation. If there are not, then the operation shall fail and set the ***returnValue*** StatusValue to STATUS_ERR_RESOURCES.

The communication state between an `XRCE Client` and an `XRCE Agent` is managed by the associated `ProxyClient`. Therefore deletion of an existing `ProxyClient` resets any prior communication state between the client and the agent. Any messages that were cached pending acknowledgments shall be discarded.

If the `Agent` creates a `ProxyClient` object it shall:

- Initialize its state to have the specified ***session_id***.
- Initialize the built-in streams with sequence number 0.
- Set the ***returnValue*** StatusValue to STATUS_OK.
- Return a representation of the XRCE Agent in the ***agent_info***.

The Agent and Client may use the ***client_timestamp*** and ***agent_timestamp*** to detect time-synchronization differences between the XRCE Client and the XRCE Agent. The use of this information is left outside the scope of this specification.

The Agent and Client may use the `XrceVersion` and `XrceVendorId` to further configure their protocol.

7.8.2.2 get_info

Inputs

- ***info_mask*** (InfoMask): selects the kind of information to retrieve.

- ***client_info*** (ObjectInfo): a representation of the `Client`.

Outputs

- ***returnValue*** (ResultStatus): indicates whether the operation succeeded and the current status of the XRCE `ProxyClient` object.
- ***agent_info*** (ObjectInfo): a representation of the `Agent`.

Both ***client_info*** and ***agent_info*** use the type `ObjectInfo` defined in Annex A, IDL Types as:

```
@extensibility(FINAL)
struct ObjectInfo {
    @optional  ActivityInfoVariant  activity;
    @optional  ObjectVariant        config;

};
```

The operation **get_info** returns information on the XRCE `Agent` and may be used prior to establishing a `Session` with the XRCE `Agent`—that is, before calling the operation **create_client** on the XRCE `Root`.

The operation **get_info** may be used over a different transport or connection, allowing a `Client` to search and discover the presence of XRCE `Agent` objects and select one (or more) with a suitable configuration and availability.

The `ObjectVariant` member within ***client_info*** shall contain a `CLIENT_Representation`, which provides information on the XRCE `Client` that makes the request. This type is defined in Annex A, IDL Types and also shown in 7.8.2.1.

The ***client_key*** field of `CLIENT_Representation` shall be set to the value `CLIENTKEY_INVALID` (see Annex A, IDL Types) in order to not unnecessarily disclose the `ClientKey`.

The `ObjectVariant` member within ***agent_info*** shall contain an `AGENT_ActivityInfo` which contains activity information on the XRCE `Agent`. This type is defined in Annex A, IDL Types and also shown in 7.8.2.1.

The `ActivityInfoVariant` member within ***agent_info*** shall contain an `AGENT_Representation`, which contains information on the XRCE `Agent`. This type is defined in Annex A, IDL Types

`ActivityInfoVariant` member ***address_seq*** shall be used to inform the XRCE `Client` of the transport addresses over which it can be reached and can receive calls to **create_client**.

The ***properties*** field of type `PropertySeq` available both in the `CLIENT_Representation` and the `AGENT_Representation` may be used to implement an authentication protocol for the XRCE `Agent`. The specific mechanism is outside the scope of this specification.

7.8.2.3 delete_client

Outputs

- ***returnValue*** (ResultStatus): indicates whether the operation succeeded and the current status of the object.

The `XRCE Agent` shall check the `ClientKey` to locate an existing XRCE: `ProxyClient`. If the object is not found the operation shall fail and ***returnValue*** StatusValue shall be set to STATUS_ERR_UNKNOWN_REFERENCE. If the object is found it shall be delete and ***returnValue*** StatusValue shall be set to STATUS_OK.

7.8.3 XRCE ProxyClient

The `XRCE ProxyClient` object represents a specific `XRCE Client` inside a concrete `XRCE Agent`. The `ProxyClient` object is identified by the `ClientKey`.

The logical operations on the `ProxyClient` are shown in Table 6.

Table 6 XRCE ProxyClient operations

create		ResultStatus
	creation_mode	CreationMode
	objectid_prefix	ObjectIdPrefix
	object_representation	ObjectVariant
update		ResultStatus
	objectid_prefix	ObjectIdPrefix
	object_representation	ObjectVariant
get_info		ResultStatus
	out: object_info	ObjectInfo
	info_mask	InfoMask
	object_id	ObjectId
delete		ResultStatus
	object_id	ObjectId

7.8.3.1 create

Inputs

- ***creation_mode*** (CreationMode): controls the behavior of the operation when there is an existing object that partially matches the description of the object that the client wants to create.
- ***objectid_prefix*** (ObjectIdPrefix): configures the desired `ObjectId` for the created object.
- ***object_representation*** (ObjectVariant): a representation of the object that the client wants to create.

Outputs

- ***returnValue*** (ResultStatus): indicates whether the operation succeeded and the current status of the object. The ***object_id*** in the ***returnValue*** shall be derived from the ***object_prefix*** input parameter.

This operation attempts to create a XRCE object according to the specification provided in the ***object_representation*** parameter. The `ObjectVariant` is a union discriminated by the `ObjectKind` that is used to define the kind of XRCE object being created, see 7.7.3. We will refer to this `ObjectKind` as the "input_objectkind".

The ***object_prefix*** parameter contains the `ObjectIdPrefix` used to determine the `ObjectId` for the object. See 7.7.6. The combination of the ***objectid_prefix*** and the `ObjectKind` contained in the ***object_representation*** discriminator shall be used to construct the "input" `ObjectId`. We shall refer to this `ObjectId` as the "***input_objectid***".

The selected member of the `ObjectVariant` contains the information required to construct an object of `ObjectKind` ***input_objectkind***.

The ***creation_mode*** affects the behavior of the create operation as specified in Table 7.

Table 7 -- CreationMode influence on create operation

creation mode **reuse**	*creation mode* **replace**	*input objectid* **exists**	***Result***
Don't care	Don't care	NO	Create object according to Table 8.
FALSE	FALSE	YES	No action taken. Set the StatusValue within ***returnValue*** to: STATUS_ERR_ALREADY_EXISTS.
FALSE	TRUE	YES	Delete existing object as specified by the **delete** operation. Create object according to Table 8. Set the StatusValue within ***returnValue*** to: STATUS_OK.
TRUE	FALSE	YES	Check if ***object_representation*** matches the existing Object: If it matches no action is taken. Set the StatusValue within ***returnValue*** to: STATUS_OK_MATCHES. If it does not match no action is taken. Set the StatusValue within ***returnValue*** to: STATUS_ERR_MISMATCH.
TRUE	TRUE	YES	Check if ***object_representation*** matches the existing Object: If it matches, no action is taken. Set the StatusValue within ***returnValue*** to: STATUS_OK_MATCHES If it does not match, delete existing object as specified by the **delete** operation and then create a new object according to Table 8. Set the StatusValue within ***returnValue*** to: STATUS_OK.

As described in 7.7.3 the `ObjectVariant` type used for the ***object_representation*** is a union type discriminated by the `ObjectKind`. However the representations for the different kinds of objects all derive from either `OBJK_Representation2_Base` or `OBJK_Representation3_Base`. Therefore they all have at least the REPRESENTATION_BY_REFERENCE and the REPRESENTATION_AS_XML_STRING. Object representations deriving `OBJK_Representation3_Base` also have a REPRESENTATION_IN_BINARY.

Certain representations support the representation of nested objects. For example, as seen in 7.7.3.6.2, the XML representation of a XRCE `DomainParticipant` may contain representations of nested `Topic`, `Publisher`, `Subscriber`, `DataWriter`, and `DataReader` objects. In this case, the creation of the XRCE object shall also create the nested objects and the failure to create any nested entity shall be considered a failure to create the contained entity as well.

Some of the XRCE objects may be defined by this specification as proxies for DDS entities. In this case the creation of the XRCE Object will automatically trigger the creation of the proxy DDS `Entity`. Failure to create a DDS `Entity` shall be considered a failure to create the proxy XRCE object as well.

If the creation of the XRCE object fails then there should be no associated DDS-RTPS discovery traffic generated by the `Agent`. This means that all DDS entities shall be created disabled, such that the creation does not result in DDS-RTPS discovery traffic, and enabled (if so configured by their QoS) only after it has been determined that the creation has succeeded.

If the creation succeeds the Agent shall set the StatusValue within ***returnStatus*** to STATUS_OK..

The creation of XRCE objects is done in accordance to the ***object_representation*** parameter. The specific behavior depends on the `ObjectKind`. See Table 8.

Table 8 Behavior of the create operation according to the ObjectKind

ObjectKind	Create behavior
OBJK_QOSPROFILE	The `ObjectVariant` is a `OBJK_QOSPROFILE_Representation` which references or contains a `QosProfile` definition. The agent shall use that definition to create a XRCE `QosProfile` in accordance to the representation defined in 7.7.3.2.
OBJK_TYPE	The `ObjectVariant` is a `OBJK_TYPE_Representation` which references or contains a Type definition. The agent shall use that definition to create a XRCE `Type` in accordance to the representation defined in 7.7.3.3.
OBJK_APPLICATION	The `ObjectVariant` is a `OBJK_APPLICATION_Representation` which references or contains XRCE Application definition. The agent shall use that definition to create a XRCE `Application` with all the contained entities found within the definition in accordance to the representation defined in 7.7.3.5.
OBJK_PARTICIPANT	The `ObjectVariant` is a `OBJK_PARTICIPANT_Representation` which references or contains a `DomainParticipant` definition. The agent shall use that definition to create a XRCE `DomainParticipant` and an associated DDS `DomainParticipant` with all the contained entities found within the definition in accordance to the representation defined in 7.7.3.6.
OBJK_TOPIC	The `ObjectVariant` is a `OBJK_TOPIC_Representation` which references or contains a `Topic` definition. The agent shall locate the XRCE `DomainParticipant` identified by the ***participant_id***. If this object is not found the operation shall fail and return STATUS_ ERR_UNKNOWN_REFERENCE. The agent shall use the definition to create a XRCE `Topic` in accordance with the representation defined in 7.7.3.7 and an associated DDS `Topic`. The DDS `Topic` shall be created using the `DomainParticipant` identified by the ***participant_id***.
OBJK_PUBLISHER	The `ObjectVariant` is a `OBJK_PUBLISHER_Representation` which references or contains a `Publisher` definition. The agent shall locate the XRCE `DomainParticipant` identified by the ***participant_id***. If this object is not found the operation shall fail and return STATUS_ ERR_UNKNOWN_REFERENCE. The agent shall use the definition to create a XRCE `Publisher` in accordance with the representation defined in 7.7.3.8 and an associated DDS `Publisher`. The DDS `Publisher` shall be created using the `DomainParticipant` identified by the ***participant_id***.
OBJK_SUBSCRIBER	The `ObjectVariant` is a `OBJK_SUBSCRIBER_Representation` which references or contains a `Subscriber` definition. The agent shall locate the XRCE `DomainParticipant` identified by the ***participant_id***. If this object is not found the operation shall fail and return STATUS_ ERR_UNKNOWN_REFERENCE. The agent shall use the definition to create a XRCE `Subscriber` in accordance with the representation defined in 7.7.3.9 and an associated DDS `Subscriber`. The DDS `Subscriber` shall be created using the `DomainParticipant` identified by the

	participant_id.
OBJK_DATAWRITER	The `ObjectVariant` is a `DATAWRITER_Representation` which references or contains a `DataWriter` definition. The agent shall locate the XRCE `Publisher` identified by the ***publisher_id***. If this object is not found the operation shall fail and return STATUS_ ERR_UNKNOWN_REFERENCE. The agent shall use the definition to create a XRCE `DataWriter` in accordance with the representation defined in 7.7.3.10 and an associated DDS `DataWriter`. The DDS `DataWriter` shall be created using the `Publisher` identified by the ***publisher_id***.
OBJK_DATEREADER	The `ObjectVariant` is a `DATAWRITER_Representation` which references or contains a `DataReader` definition. The agent shall locate the XRCE `Subscriber` identified by the ***subscriber_id***. If this object is not found the operation shall fail and return STATUS_ ERR_UNKNOWN_REFERENCE. The agent shall use the definition to create a XRCE `DataReader` in accordance with the representation defined in 7.7.3.11 and an associated DDS `DataReader`. The DDS `DataReader` shall be created using the `Subscriber` identified by the ***subscriber_id***.

7.8.3.2 update

Inputs

- ***object_id*** (ObjectId): the object being updated.
- ***object_representation*** (ObjectVariant): of the updated object.

Outputs

- ***returnValue*** (ResultStatus): indicates whether the operation succeeded and the current status of the object.

This operation shall attempt to update an existing object in the `XRCE Agent`. If the object exists and the update is successful STATUS_OK shall be returned, otherwise a status indicating an error shall be returned:

- If the object does not already exist STATUS_ERR_UNKNOWN_REFERENCE shall be returned.
- If the update was unsuccessful due to invalid parameters, STATUS_ERR_INVALID_DATA shall be returned. If an update is unsuccessful the referenced object shall return to its previous configuration.
- If the object cannot be updated due to permission restrictions, STATUS_ERR_DENIED shall be returned.

7.8.3.3 get_info

Inputs

- ***objectid_id*** (ObjectId): the object queried.
- ***info_mask*** (InfoMask): selects the kind of information to retrieve.

Outputs

- ***returnValue*** (ResultStatus): indicates whether the operation succeeded.
- ***object_info*** (ObjectInfo): contains the current activity and configuration of the specified object.

This operation returns the configuration and activity data for an existing object.

- If the object does not already exist STATUS_ERR_UNKNOWN_REFERENCE shall be returned.

- If the object cannot be accessed due to permission restrictions STATUS_ERR_DENIED shall be returned.

7.8.3.4 delete

Inputs

- ***object_id*** (ObjectIdPrefix): the object being deleted.

Outputs

- ***returnValue*** (ResultStatus): indicates whether the operation succeeded.

This operation deletes an existing object. If the object is successfully deleted STATUS_OK shall be returned.

- If the object does not exist STATUS_ERR_UNKNOWN_REFERENCE shall be returned.
- If the object cannot be deleted due to permission restrictions, STATUS_ERR_DENIED shall be returned.

7.8.4 XRCE DataWriter

The operations are defined in Table 9.

Table 9 XRCE DataWriter operations

write		ResultStatus
	object_id	ObjectId
	data	DataRepresentation

7.8.4.1 write

Inputs

- ***object_id*** (ObjectId): the object that shall publish the data.
- ***data*** (DataRepresentation): data to be written.

Outputs

- ***returnValue*** (ResultStatus): indicates whether the operation succeeded and the current status of the object. The ***object_id*** in the ***returnValue*** shall be set to match the ***object_id*** input parameter.

This operation writes one or more samples using the XRCE DataWriter identified by the ***object_id***.

- If the data is successfully written STATUS_OK shall be returned.
- If the XRCE DataWriter object identified by the ***object_id*** does not exist, the ResultStatus STATUS_ERR_UNKNOWN_REFERENCE shall be returned.
- If the client is not allowed to write data using the referenced ***object_id*** due to permission restrictions, the ResultStatus STATUS_ERR_DENIED shall be returned.
- If the data could not be written successfully due, for example invalid data format, the ResultStatus STATUS_ERR_INVALID_DATA shall be returned.

The `DataRepresentation` type (see 7.7.2) supports multiple DataFormats. This allows sending single data items (FORMAT_DATA) as well as sequences (batches) of data items (FORMAT_SAMPLE_SEQ).

The `DataRepresentation` type also supports sending sample information in addition to the data. This is encoded in the `SampleInfo` type (see 7.7.1) allowing sending timestamps and also notifications of dispose and unregister.

If the DataRepresentation contains a Sample where the SampleInfo has the "dispose" flag set, the XRCE Agent shall call the **dispose** operation on the corresponding DDS DataWriter for the instance identified in the associated data. Similarly there is a Sample where the SampleInfo has the "unregister" flag set, the XRCE Agent shall call the **unregister** operation on the corresponding DDS DataWriter for the instance identified in the associated data.

7.8.5 XRCE DataReader

The operations are defined in Table 10 .

Table 10 XRCE DataReader operations

read		ResultStatus
	out: read_data	DataRepresentation
	object_id	ObjectId
	read_specification	ReadSpecification

7.8.5.1 read

Inputs

- ***object_id*** (ObjectId): the object to read data from.
- ***read_specification*** (ReadSpecification): the operation will only return data that matches the constraint.

Outputs

- ***returnValue*** (ResultStatus): indicates whether the operation succeeded.
- ***data_read*** (DataRepresentation): data matching the ***read_spec*** or nil if there was an error.

This operation reads one or more samples from the XRCE DataReader identified by the ***object_id***. If the data is successfully read STATUS_OK shall be returned.

- If the object does not exist STATUS_ERR_UNKNOWN_REFERENCE shall be returned.
- If the client is not allowed to read data using the referenced ***object_id*** due to permission restrictions, STATUS_ERR_DENIED shall be returned.

The ***read_spec*** parameter controls the data returned by this operation. The fields of this structure shall be interpreted as described in Table 11.

Table 11 Interpretation of the ReadSpecification

field	*type*	*interpretation*
data_format	DataFormat	Selects one the data formats. See 7.7.1
content_filter_expression	string	A content filter expression selecting which data to read. The syntax shall be as specified in Annex B (Syntax for Queries and Filters) of the DDS specification [DDS].
max_samples (DataDeliveryControl)	unsigned short	Maximum number of samples to return as a result of the read. The special value MAX_SAMPLES_ZERO =0 is used to cancel any

		prior read operation that may still be active. The special value MAX_SAMPLES_UNLIMITED =0xffff is used to indicate that there is limit on the number of samples returned.
max_elapsed_time (DataDeliveryControl)	unsigned short	Maximum amount of time in seconds that may be spent delivering the samples from the read operation. The units are seconds from the time the call is made. The special value MAX_ELAPSED_TIME_UNLIMITED = 0 indicates there is no maximum and the operation shall continue until some other condition is met or the operation is explicitly cancelled.
max_bytes_per_sec (DataDeliveryControl)	unsigned short	Maximum rate in bytes per second at which the data may be returned to the read operation.
min_pace_period (DataDeliveryControl)	unsigned short	Minimum separation between data messages returned from the read operation in milliseconds.

The setting of the ***data_format*** controls whether the read operation returns a single sample per message or a collection of samples. It also determines whether the data or it includes the additional information that appears in the `SampleInfo` (see Annex A IDL Types). The additional information contains sequence numbers and time stamps.

The setting of the ***content_filter_expression*** configures a content filter that is applied to the samples in the DataReader cache. Only samples for which the filter evaluates to TRUE shall be returned to the XRCE Client.

The setting of the ***max_samples*** configures the read operation to terminate after the specified number of samples has been returned. The value MAX_SAMPLES_ZERO can be used to cancel the currently active read operation without sending any more samples. The value MAX_SAMPLES_UNLIMITED indicates there is no limit to the number of samples returned.

The setting of the ***max_elapsed_time*** configures the read operation to terminate after the specified time has elapsed from the moment the read operation was made. The value MAX_ELAPSED_TIME_UNLIMITED indicates that there is no termination condition based on the elapsed time.

The setting of the ***max_bytes_per_sec*** configures the maximum rate in bytes per second at which samples may be returned.

The setting of the ***min_pace_period*** configures the minimum interval in milliseconds between the sample messages sent from the Agent to the Client. This period makes it possible for the client to go into a sleep cycle between messages.

8 XRCE Protocol

8.1 General

The XRCE `Agent` implements the operations specified in the DDS-XRCE Object Model that are driven by messages between the XRCE `Client` and XRCE `Agent`. The DDS-XRCE message protocol is designed specifically to address the limited CPU, power, and network bandwidth found in many types of low-powered devices and to enable the device to be discoverable in the larger DDS network. Specifically, it is designed to meet the unique challenges posed by these types of devices. The main features include:

- Operate over networks with bandwidth limited to 40-100Kbps.
- Work with devices that undergo sleep cycles. These devices may be active once every few minutes, days, months, or even years.
- Be simple and programming-language independent, supporting devices that are programmed in a highly specialized language or frameworks.
- Support a minimal discovery protocol, allowing plug-and-play deployments where the Agent location is dynamically discovered.
- Support accessing the full capabilities of DDS. Any data type can be published or subscribed to with any DDS QoS.
- Support sending updates to multiple data-times on the same or multiple DDS Topics efficiently.
- Support receiving information both reliably and in a best effort manner, even if the information was sent while the Client was undergoing a sleep cycle.
- Support secure communication at the transport level.
- Provide full read/write access to any data in the DDS Global Data Space (subject to access control limits).
- Provide a full implementation requiring less than 100KB of code.

In contrast to applications that use the DDS API directly, XRCE `Clients`:

- Do not have a standard API, so they are not portable across vendor implementations.
- Cannot operate without infrastructure support. They need a XRCE `Agent` to be reachable to them. This is a necessary consequence of the need for XRCE `Clients` to undergo deep sleep cycles.
- Do not communicate directly peer-to-peer. All communications are brokered (relayed) by one or more DDS-XRCE Agents. This is also a necessary consequence of the need for Clients to undergo deep sleep cycles.

8.2 Definitions

XRCE `Clients` and XRCE `Agents` exchange messages to execute operations on the XRCE Agent and return results. The DDS-XRCE Protocol uses the terms **client**, **agent**, **session,** and **message** defined in the subclauses below.

At a high level, a **client** communicates with an **agent** using the DDS-XRCE protocol, exchanging **messages** on a **stream** belonging to a **session**.

8.2.1 Message

A **message** is the unit of information sent via the transport and is a structured sequence of bytes sent on a DDS-XRCE transport. A **message** has a sequence number that is used for ordering of messages, or for identifying messages that have been dropped by the transport.

The underlying XRCE Transport shall transfer each **message** as a unit. A single XRCE Transport “message” shall transport a single XRCE **message**.

XRCE messages shall be encoded assuming the first byte has a 16-byte alignment. Therefore the encoding is independent of any transport heading or prefix that may precede it.

8.2.2 Session

A **session** defines a bi-directional connection between a **client** and an **agent** that has been established with a handshake. The **session** is needed to exchange messages with the XRCE **agent**. An XRCE **client** may send messages over multiple **sessions,** for example if it communicates with multiple XRCE **agents**.

A **session** can contain independent, reliable, and best-effort message **streams**. Each **session** may have up to 256 streams.

There can be at most one active **session** between an XRCE **client** and an XRCE **agent**. Creation of a new **session** closes any previous sessions.

8.2.3 Stream

A **stream** represents an independent ordered flow of messages within a **session**. Messages are ordered within a **stream** by means of a sequence number. The sequence numbers used by different streams are independent of each other.

Streams can be reliable or best efforts. Each **stream** uses a constant endianness to encode the data in the message/submessage headers and payload.

8.2.4 Client

An XRCE **client** is the entity that initiates the establishment of a session with an XRCE **agent**. An XRCE **client** may send and receive messages to the **agent** on streams belonging to an established XRCE **session**.

8.2.5 Agent

An XRCE **agent** is the entity that listens to and accepts requests to establish **sessions** from XRCE **clients**. An XRCE **agent** may send and receive messages to a **client** on **streams** belonging to an established **session**.

8.3 Message Structure

8.3.1 General

An XRCE **message** is composed of a message `Header` followed by one or more `Submessages` and shall be transferred as a unit by the underlying XRCE Transport.

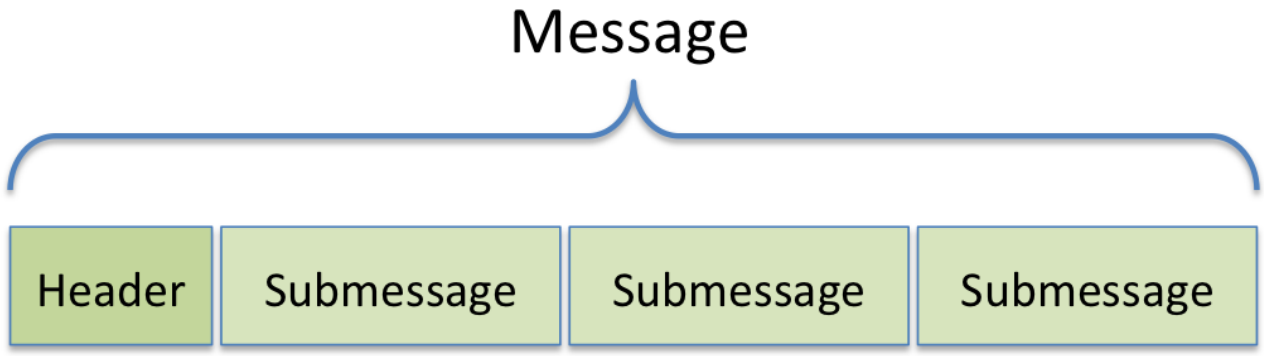
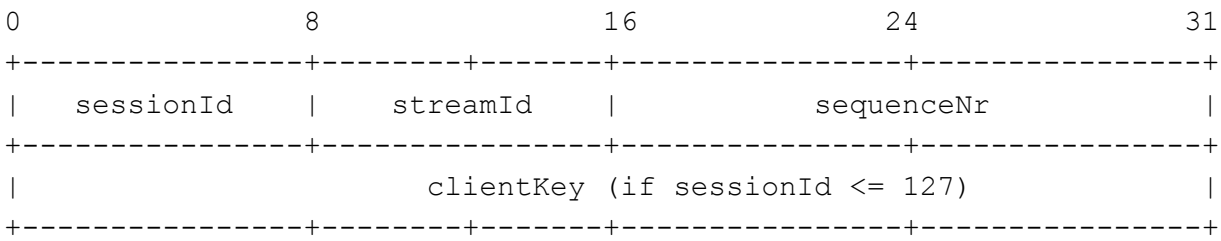


Figure 6 — Message structure

8.3.2 Message Header

The header is structured as follows:



8.3.2.1 Sessions and the sessionId

An XRCE **session** is established between the `XRCE Client` and `XRCE Agent` to establish an initial context for the communications. This includes the exchange of protocol versions, vendor identification, and other information needed to correctly process messages.

A **session** is identified by an 8-bit ***sessionId***. The ***sessionId*** is unique to an `XRCE Agent` for a given `XRCE Client`. The ***sessionId*** also determines whether the `Header` includes a ***clientKey*** or not.

- If the ***sessionId*** is between 0 and 127 (0x00 to 0x7f), both included, then the `Header` **shall** include the ***clientKey*** and the ***sessionId*** is scoped by the ***clientKey***.
- If the ***sessionId*** is between 128 and 255 (0x80 to 0xff), both included, then the `Header` **shall not** include the ***clientKey*** and the ***sessionId*** is scoped by the source address of the message.

If the ***clientKey*** does not appear explicitly in the message header, the `XRCE Agent` must be able to locate it from the source address of the message (see clause 8.3.2.4).

The following two values of the ***sessionId*** are reserved:

- The value 0 (0x00) shall be used to indicate the lack of a **session** within a `Header` containing a ***clientKey***. This value is referred to as SESSION_ID_NONE_WITH_CLIENT_KEY.
- The value 128 (0x80) shall be used to indicate the lack of a **session** within a `Header` that does not contain a ***clientKey***. This value is referred to as SESSION_ID_NONE_WITHOUT_CLIENT_KEY.

8.3.2.2 Streams and the streamId

An XRCE **stream** represents an independent flow of information between a XRCE Client and a XRCE Agent. Each XRCE **message** belongs to a single **stream**. Messages belonging to the same **stream** must be delivered in the order they are sent. Messages belonging to different streams are not ordered relative to each other.

Streams are scoped by the **session** they belong to.

The ***streamId*** with value 0 (0x00) is referred as STREAMID_NONE. This stream is used for messages exclusively containing submessages that do not belong to any stream.

The streams with ***streamId*** between 1 (0x01) and 127 (0x7F), both included, shall be best-effort streams.

The streams with ***streamId*** between 128 (0x80) and 255 (0xFF), both included, shall be reliable streams.

Based on the rules above if the ***streamId*** is not STREAMID_NONE, then the leading bit of the ***streamId*** can be interpreted as a flag that indicates the reliability of the stream.

There are two built-in streams that are created whenever a **session** is created:

- A built-in best-effort **stream** identified by a ***streamId*** with value 1 (0x01). This is referred to as STREAMID_BUILTIN_BEST_EFFORTS.

- A built-in reliable **stream** identified by a ***streamId*** with value 128 (0x80). This is referred to as STREAMID_BUILTIN_RELIABLE.

8.3.2.3 sequenceNr

The ***sequenceNr*** is used to order messages within a **stream** and it is scoped to the **stream**. Messages belonging to different streams are unordered relative to each other:

- For the **stream** with ***streamId*** STREAMID_NONE, the ***sequenceNr*** does not impose any order; however it still may be used to discard duplicate messages.
- For the **stream** with ***streamId*** different from STREAMID_NONE, the ***sequenceNr*** imposes an order. Messages within a **stream** shall not be delivered out of order. In addition duplicate messages shall be discarded.

Addition and comparison of sequence numbers shall use Serial Number Arithmetic as defined by [IETF RFC-1982] with SERIAL_BITS set to 16. This implies that the maximum number of outstanding (unacknowledged) messages for a specific client session **stream** is limited to 2^{15}—that is, 32768.

The ***sequenceNr*** shall be encoded using little endian format.

8.3.2.4 clientKey

The ***clientKey*** uniquely identifies and authenticates an XRCE Client to the XRCE Agent.

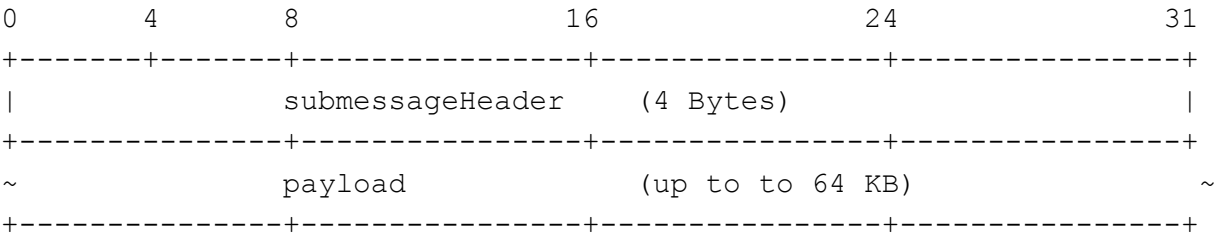
The ***clientKey*** shall be present on the Header if the ***sessionId*** is between 0 and 127. See clause 8.3.2.1:

- If the ***clientKey*** is present, it shall contain the `ClientKey` associated with the XRCE Client.
- If the ***clientKey*** is not present, the XRCE Agent shall be able to derive the `ClientKey` associated with the XRCE Client from the source address of the message. This means that the `ClientKey` has either been pre-configured on the XRCE Agent for that particular source address, or it has been exchanged as part of the session establishment. See clause 7.8.2.1.

Any exchange of the ***clientKey*** is protected by the security mechanisms provided by the XRCE transport. These security mechanisms are transport-specific and may involve a pairing of each device with the agent or some initial handshake used to establish a secure transport connection. The specific transport security mechanisms are outside the scope of this specification.

8.3.3 Submessage Structure

Following the message header there shall be one or more submessages. A `Submessage` shall be composed of a `SubmessageHeader` and a payload.



The ability to place multiple Submessages within a single message reduces bandwidth by enabling multiple resources to be operated on with a single message.

Submessages shall start at an offset that is a multiple of 4 relative to the beginning of the Message. This means that additional padding may be added between the end of a submessage and the beginning of the next submessage.

8.3.4 Submessage Header

Every `Submessage` shall start with a `SubmessageHeader`. The `SubmessageHeader` shall be structured as follows:

```
0       4       8               16              24              31
+-------+-------+---------------+---------------+---------------+
| submessageId  |      flags    |       submessageLength        |
+-------+-------+---------------+---------------+---------------+
```

8.3.4.1 submessageId

The ***submessageId*** identifies the kind of submessage. The kinds of submessages are defined in 8.3.5.

8.3.4.2 flags

The ***flags*** field contains information about the content of the Submessage.

Bit 0, the 'Endianness' bit, shall indicate the endianness used to encode the submessage header and payload. If the Endianness bit is set to 0, the encoding shall be big endian and otherwise little endian.

The ***flags*** field for all submessage kinds shall have the Endianness bit. Specific submessage kinds may define additional flag bits.

8.3.4.3 submessageLength

The ***submessageLength*** indicates the length of the Submessage (excluding the Submessage header).

The ***submessageLength*** shall be encoded using little endian format, independent of the value of the *flags*.

8.3.4.4 payload

The ***payload*** contains information specific to the submessage whose format depends on the kind of submessage identified by the ***submessageId***.

The definition of the ***payload*** shall use the data types defined in clause 7.7. See clause 8.3.5 and its subclauses.

8.3.5 Submessage Types

DDS-XRCE defines the 13 kinds of Submessages shown in the figure below:

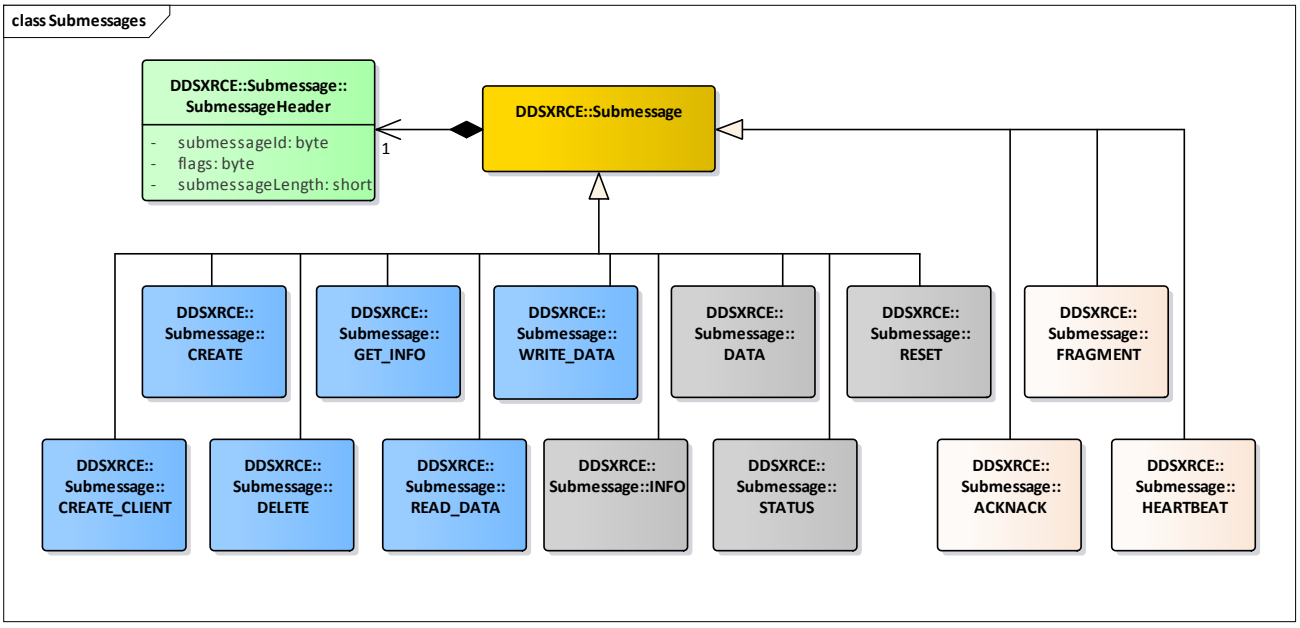


Figure 7 — DDS-XRCE submessages

Each submessage is identified by the ***submessageId***. Some submessages may only be sent in one direction (e.g. only XRCE `Client` to XRCE `Agent` or only XRCE `Agent` to XRCE `Client`) whereas others are bi-directional.

Table 12 – List of SubmesageId values and their purpose

SubmessageId	Value	Purpose
CREATE_CLIENT	0	Client to Agent. Initiates the connection between Client and Agent. Creates a ProxyClient on the Agent. Causes the Agent to call the `Root`::**create_client** operation.
CREATE	1	Client to Agent. Creates an XRCE Object. Causes the Agent to call the `ProxyClient`::**create** operation.
GET_INFO	2	Client to Agent. Requests information on an XRCE Object. Causes the Agent to call the operation `Root`::**get_info** or `ProxyClient`::**get_info**.
DELETE	3	Client to Agent. Deletes an object or set of XRCE Objects. Causes the Agent to call the `ProxyClient`::**delete** operation or the `Root`::**delete_client** operation.
STATUS_AGENT	4	Agent to Client. Sent in response to CREATE_CLIENT. Contains information about the Agent. Carries the return value of the `Root`::**create_client** operation.
STATUS	5	Agent to Client; typically in response to CREATE, UPDATE or DELETE. Contains information about the status of an Xrce object. Carries the return value of the `ProxyClient`::**create**, **update**, or **delete** operations.
INFO	6	Agent to Client. Typically sent in response to a GET_INFO. Contains detailed information about an Xrce: Object or the XRCE Agent. Carries the return value of the operation `Root`::**get_info** or `ProxyClient`::**get_info**
WRITE_DATA	7	Client to Agent. Used to write data using a XRCE DataWriter. Causes the Agent to call the `ProxyClient`::**write** operation.
READ_DATA	8	Client to Agent. Used to read data using a XRCE DataReader. Causes the Agent to call the `ProxyClient`::**read** operation.
DATA	9	Agent to Client in response to a READ_DATA provides data received by a XRCE DataReader. Carries the return value of the `ProxyClient`::**read** operation.
ACKNACK	10	Bi-directional. Sends a positive and/or negative acknowledgment to a range of sequence numbers.

HEARTBEAT	11	Bi-directional. Informs of the available sequence number ranges.
RESET	12	Bi-directional. Resets a session.
FRAGMENT	13	Bi-directional. Communicates a data fragment. Used to send messages of size larger than what is supported by the underlying transport.

8.3.5.1 CREATE_CLIENT

The **CREATE_CLIENT** submessage shall be sent by the XRCE `Client` to create a XRCE `ProxyClient`.

Reception of this submessage shall result in the XRCE `Agent` calling the **create_client** operation on the XRCE `Root` object, see 7.8.2.1. The parameters to this operation are obtained from the *payload*.

The XRCE Agent shall send a **STATUS_AGENT** message in response, see 0.

8.3.5.1.1 flags

The **CREATE_CLIENT** submessage does not define any additional flag bits beyond the common ones specified in 8.3.4.2.

8.3.5.1.2 payload

The payload shall contain the XCDR representation of the `CREATE_CLIENT_Payload` object defined in Annex A IDL Types as:

```
@extensibility(FINAL)
struct CLIENT_Representation {
    XrceCookie   xrce_cookie;  // XRCE_COOKIE
    XrceVersion  xrce_version;
    XrceVendorId xrce_vendor_id;
    Time_t       client_timestamp;
    ClientKey    client_key;
    SessionId    session_id;
    @optional  PropertySeq properties;
};

@extensibility(FINAL)
struct CREATE_CLIENT_Payload : BaseObjectRequest {
    CLIENT_Representation  client_representation;
};
```

The payload contains the ***client_representation*** input parameter to the **create_client** call.

8.3.5.2 CREATE

The **CREATE** submessage shall be sent by the XRCE `Client` to create a XRCE `Object`. An example is creating an XRCE:`DataWriter` with a QoS profile.

Reception of this submessage shall result in the XRCE Agent calling the **create** operation on the XRCE ProxyClient object, see 7.8.3.1. The parameters to this operation shall be obtained from the `SubmessageHeader` flags and *payload*.

The XRCE Agent shall send a **STATUS** submessage in response, see 8.3.5.6.

8.3.5.2.1 flags

The **CREATE** submessage defines two additional flag bits that encode the ***creation_mode*** input parameter to the **create** call:

Bit 1, the 'Reuse' bit, encodes the value of the `CreationMode` ***reuse*** field.

Bit 2, the 'Replace' bit, encodes the value of the `CreationMode` ***replace*** field.

These flag bits modify the behavior of the XRCE Agent receiving the **CREATE** message. See clause 7.8.3.1.

8.3.5.2.2 payload

The payload shall contain the XCDR representation of the `CREATE_Payload` object defined in Annex A IDL Types and also shown below. See also 7.7.3 for the definition and interpretation of the `ObjectVariant`:

```
@extensibility(FINAL)
struct CREATE_Payload : BaseObjectRequest {
    ObjectVariant object_representation;
};
```

The payload derives from `BaseObjectRequest`, which contains the ***object_id*** parameter to the **create** call.

The payload contains the ***object_representation*** input parameter to the **create** call.

8.3.5.3 GET_INFO

The **GET_INFO** submessage shall be sent by the XRCE `Client` to get information about a resource identified by its ***object_id***.

Reception of this submessage shall result in the XRCE Agent calling the **get_info.** The targeted XRCE `Object` shall depend on the `ObjectKind` encoded in the last 4 bits of the ***object_id.***

- If the `ObjectKind` is set to OBJK_AGENT, then it shall result in the XRCE Agent calling the **get_info** operation on the XRCE `Root` object (see 7.8.3.3).
- If the `ObjectKind` is set to one of OBJK_PARTICIPANT, OBJK, OBJK_PUBLISHER, OBJK_SUBSCRIBER, OBJK_DATAWRITER, OBJK_DATAREADER, OBJK_TYPE, OBJK_QOSPROFILE, or OBJK_APPLICATION. That is to a value between 0x01 and 0x0c (both included), then it shall result in the XRCE Agent calling the **get_info** operation on the XRCE `ProxyClient` object (see 7.8.3.3).

The parameters to this operation shall be obtained from the *payload*.

The XRCE `Agent` shall send an **INFO** submessage in response to this message, see 8.3.5.6.

8.3.5.3.1 flags

The **GET_INFO** submessage does not define any additional flag bits beyond the common ones specified in 8.3.4.2.

8.3.5.3.2 payload

The payload shall contain the XCDR representation of the `GET_INFO_Payload` object defined in Annex A IDL Types as:

```
bitmask InfoMask {
    @position(0) INFO_CONFIGURATION,
    @position(1) INFO_ACTIVITY
};
@extensibility(FINAL)
struct GET_INFO_Payload : BaseObjectRequest {
    InfoMask  info_mask;
};
```

The payload derives from `BaseObjectRequest`, which contains the ***object_id*** parameter to the **get_info** call.

The payload also contains the ***info_mask*** input parameter to the **get_info** call.

8.3.5.4 DELETE

The **DELETE** submessage shall be sent by the XRCE `Client` to delete the XRCE:`ProxyClient` or any other XRCE `Object` (e.g. XRCE:DataWriter).

Reception of this submessage shall result in the XRCE Agent calling either the `delete_client` operation on the XRCE `Root` (see 7.8.2.3), or else the `delete` operation on the XRCE `ProxyClient` object (see 7.8.3.4).

The related XRCE Object is identified by the ***object_id*** field in the *payload*.

If the `ObjectVariant` contained within the *payload* has `ObjectKind` set to OBJK_CLIENT, then the XRCE Agent shall call the `delete_client` operation. Otherwise it shall call the `delete` operation.

The parameters to the `delete_client` or the `delete` operation shall be obtained from the *payload*.

The XRCE Agent shall send a **STATUS** submessage in response, see 8.3.5.6.

8.3.5.4.1 flags

The **DELETE** submessage does not define any additional flag bits beyond the common ones specified in 8.3.4.2.

8.3.5.4.2 payload

The payload shall contain the XCDR representation of the `DELETE_Payload` object defined in Annex A IDL Types as:

```
@extensibility(FINAL)
struct DELETE_Payload : BaseObjectRequest {
};
```

The payload derives from `BaseObjectRequest` which contains the ***object_id*** that identifies the XRCE Object to delete.

8.3.5.5 STATUS_AGENT

The **STATUS_AGENT** submessage shall be sent by the XRCE `Agent` in response to a **CREATE_CLIENT** submessage.

The submessage shall contain the ***returnStatus*** to the **create_client** operation invocation that was triggered by the reception of the corresponding **CREATE_CLIENT** message.

8.3.5.5.1 flags

The **STATUS_AGENT** submessage does not define any additional flag bits beyond the common ones specified in 8.3.4.2.

8.3.5.5.2 payload

The payload shall contain the XCDR representation of the `STATUS_AGENT_Payload` object defined in Annex A IDL Types as:

```
@extensibility(FINAL)
struct AGENT_Representation {
    xrce_cookie;  // XRCE_COOKIE
    XrceVersion  xrce_version;
    XrceVendorId xrce_vendor_id;
    Time_t       agent_timestamp;
    @optional  PropertySeq  properties;
};

@extensibility(FINAL)
struct STATUS_AGENT_Payload : BaseObjectReply {
    AGENT_Representation agent_info;
};
```

If the operation fails, the `STATUS_AGENT_Payload` shall have the `ResultStatus` within the `BaseObjectReply` set to with the `StatusValue` that corresponds to the type of error encountered. Otherwise, it shall have it set to STATUS_OK.

The ***request_id*** and ***object_id*** within the `BaseObjectReply` shall match the namesake fields in the `BaseObjectRequest` of the corresponding **CREATE_CLIENT** message.

The ***xrce_cookie*** shall be set to the four bytes {'X', 'R', 'C', 'E'}.

The ***xrce_version*** shall be set to the version of the XRCE protocol that the `Agent` will implement in its connection to the `Client`.

8.3.5.6 STATUS

The **STATUS** submessage shall be sent by the XRCE `Agent` in response to a **CREATE** or **DELETE.**

The **STATUS** submessage shall also be sent by the XRCE `Agent` in response to a **READ_DATA** submessage when the ***returnStatus*** to the **read_data** operation is anything other than `STATUS_OK`.

The **STATUS** submessage shall contain the ***returnStatus*** to the operation that was triggered by the corresponding request message. For example, if the request message was a **CREATE**, the **STATUS** payload shall contain the ***returnStatus*** to the **create** operation.

8.3.5.6.1 flags

The **STATUS** submessage does not define any additional flag bits beyond the common ones specified in 8.3.4.2.

8.3.5.6.2 payload

The payload shall contain the XCDR representation of the `STATUS_Payload` object defined in Annex A IDL Types as:

```
@extensibility(FINAL)
struct STATUS_Payload : BaseObjectReply {
};
```

If the operation fails, the `ResultStatus` within the `BaseObjectReply` shall be set to the `StatusValue` that corresponds to the type of error encountered. Otherwise, it shall have it set to STATUS_OK.

The ***request_id*** and ***object_id*** within the `BaseObjectReply` shall match the namesake fields in the corresponding request message.

8.3.5.7 INFO

The **INFO** submessage shall be sent by the XRCE `Agent` to the XRCE `Client` in response to a **GET_INFO** message.

The submessage contains the ***returnStatus*** and output parameters of the `get_info` operation that was triggered by the corresponding request message.

8.3.5.7.1 flags

The **INFO** submessage does not define any additional flag bits beyond the common ones specified in 8.3.4.2.

8.3.5.7.2 payload

The payload shall contain the XCDR representation of the `INFO_Payload` object defined in Annex A IDL Types. See also clause 7.7.13 for a description of the ObjectInfo contained in the payload.

```
@extensibility(FINAL)
struct ObjectInfo {
    @optional ActivityInfoVariant  activity;
    @optional ObjectVariant        config;
};

@extensibility(FINAL)
struct INFO_Payload : BaseObjectReply {
    ObjectInfo     object_info;
};
```

If the operation fails the `ResultStatus` within the `BaseObjectReply` shall be set to the `StatusValue` that corresponds to the type of error encountered. Otherwise it shall have it set to STATUS_OK.

The ***request_id*** and ***object_id*** within the `BaseObjectReply` shall match the identically named fields in the `BaseObjectRequest` of the corresponding GET_INFO message.

The ***activity*** and ***config*** within members within the `INFO_Payload` shall contain the value of the identically named output parameters of the `get_info` operation.

8.3.5.8 WRITE_DATA

The **WRITE_DATA** submessage is used by the XRCE `Client` to write data using a XRCE `DataWriter` object within the XRCE `Agent`.

Reception of this submessage shall result in the XRCE Agent calling the **write** operation on a XRCE `DataWriter` object (see 7.8.4.1). The XRCE Agent shall respond with a **STATUS** submessage.

The ***data*** parameter to the **write** operation shall be obtained from the *payload*.

The related XRCE `DataWriter` is identified by the ***object_id*** field in the *payload*.

Upon reception of this message the XRCE `Agent` shall located the XRCE `DataWriter` identified by the ***object_id*** and use it to write the data to the DDS domain.

8.3.5.8.1 flags

The **WRITE_DATA** sub-message uses the lowest order 4 bits of the ***flags***:

- Bit 0 indicates the 'Endianness' as specified in 8.3.4.2.
- Bits 1, 2, and 3 shall be set to indicate the `DataFormat` used for the payload. The possible values are as indicated in Table 13 below.

Table 13 – Flag bits used by the WRITE_DATA and DATA submessages

DataFormat	*Lowest order 4 bits of flags. Bit 0 encodes the Endianness*	
	Big Endian	***Little Endian***
`FORMAT_DATA`	0000 = 0x0	0001 = 0x1
`FORMAT_SAMPLE`	0010 = 0x2	0011 = 0x3
`FORMAT_DATA_SEQ`	1000 = 0x8	1001 = 0x9
`FORMAT_SAMPLE_SEQ`	1010 = 0xA	1011 = 0xB
`FORMAT_PACKED_SAMPLES`	1110 = 0xE	1111 = 0xF

For example, if the payload of the **WRITE_DATA** message uses `FORMAT_DATA_SEQ` and is encoded as Little Endian, the corresponding 8-bit options would be set to binary 00001001, hexadecimal 0x09. The lowest order bit (bit 0) is set to 1 to indicate Little Endian encoding, and bits 1-3 are set to 0, 0, and 1, respectively, to indicate `FORMAT_DATA_SEQ`.

8.3.5.8.2 payload

The format the ***payload*** depends on the `DataFormat` encoded in the ***flags*** (see 8.3.5.8.1). The correspondence shall be as shown in Table 14 below.

Table 14 – Payload format associated with each DataFormat

DataFormat	*Contents of payload.* **See Annex A** IDL Types **for the definition**
FORMAT_DATA	struct WRITE_DATA_Payload_Data
FORMAT_SAMPLE	struct WRITE_DATA_Payload_Sample
FORMAT_DATA_SEQ	struct WRITE_DATA_Payload_DataSeq
FORMAT_SAMPLE_SEQ	struct WRITE_DATA_Payload_SampleSeq
FORMAT_PACKED_SAMPLES	struct WRITE_DATA_Payload_PackedSamples

The types referenced shall be as defined in Annex A IDL Types. All the **WRITE_DATA** payload representations extend BaseObjectRequest:

```
@extensibility(FINAL)
struct SampleData {
    XCDRSerializedBuffer serialized_data;
};
@extensibility(FINAL)
struct Sample {
    SampleInfo   info;
    SampleData   data;
};

@extensibility(FINAL)
struct WRITE_DATA_Payload_Data : BaseObjectRequest {
    SampleData              data;
};

@extensibility(FINAL)
struct WRITE_DATA_Payload_Sample : BaseObjectRequest {
    Sample                sample;
};

@extensibility(FINAL)
struct WRITE_DATA_Payload_DataSeq : BaseObjectRequest {
    sequence<SampleData>    data_seq;
};
```

```
@extensibility(FINAL)
struct WRITE_DATA_Payload_SampleSeq : BaseObjectRequest {
    sequence<Sample>        sample_seq;
};

@extensibility(FINAL)
struct WRITE_DATA_Payload_PackedSamples : BaseObjectRequest {
    PackedSamples           packed_samples;
};
```

8.3.5.9 READ_DATA

The **READ_DATA** submessage is used by the XRCE `Client` to initiate a reception (read) of data from a XRCE `DataReader` object within the XRCE Agent.

Reception of this submessage shall result in the XRCE `Agent` calling the **read** operation on a XRCE `DataReader` object (see 7.8.5.1) one or more times. Depending on the ***returnStatus***, the XRCE `Agent` may respond with a **DATA** submessages or a **STATUS** submessage.

The ***read_specification*** parameters to the **read** operation shall be obtained from the *payload*.

The *payload* also configures whether there is a single or multiple calls to the read operation.

The XRCE Agent shall send one or more **DATA** submessages in response to this message, see 8.3.5.10.

The related XRCE `DataReader` is identified by the ***object_id*** field in the *payload*.

After reception of this message, the XRCE Agent shall continue to send **DATA** submessages to the client until either the "end criteria" specified in the *payload* ***read_specification*** and ***continuous_read_options*** attained or else a new **READ_DATA** message for the same ***object_id*** is received from the XRCE Client.

The **read** operation also allows a XRCE Client to control when data may be sent by the XRCE Agent so that the `Agent` does not unnecessarily wake up the `Client` during its sleep cycle.

8.3.5.9.1 flags

The **READ_DATA** submessage does not define any additional flag bits beyond the common ones specified in 8.3.4.2.

8.3.5.9.2 payload

The payload shall contain the XCDR representation of the `READ_DATA_Payload` object defined in Annex A IDL Types as:

```
@extensibility(APPENDABLE)
struct DataDeliveryControl {
    unsigned short max_samples;
    unsigned short max_elapsed_time;
    unsigned short max_bytes_per_second;
```

```
    unsigned short min_pace_period;  // milliseconds
};
@extensibility(FINAL)
struct ReadSpecification  {
    DataFormat data_format;
    @optional string    content_filter_expression;
    @optional DataDeliveryControl delivery_control;
};

@extensibility(FINAL)
struct READ_DATA_Payload : BaseObjectRequest {
    ReadSpecification                read_specification;
};
```

The ***payload*** derives from `BaseObjectRequest` which contains the ***object_id*** parameter to the **read** call.

The ***payload*** also contains the ***read_specification*** input parameter to the **read** call.

The ***max_samples*** may take two special values:

- The value MAX_SAMPLES_ZERO shall be used to cancel the currently active **read** operation without sending any more samples.
- The value MAX_SAMPLES_UNLIMITED indicates there is no limit in the number of samples returned from a single call to the **read** operation.

The setting of the ***max_bytes_per_sec*** configures the maximum rate at which **DATA** messages may be returned.

The optional member ***continuous_read_options*** configures whether the `Agent` will perform one or multiple **read** calls:

- If the ***continuous_read_options*** member is not present, then the `Agent` shall call the **read** operation just once. As a result the only data returned will be the one already in the DDS `DataReader` cache.
- If the ***continuous_read_options*** member is present, then the `Agent` shall call the **read** operation multiple times. The period of calling shall be no faster than the ***pace_period***. As a result the data returned may contain data that arrives to the DDS `DataReader` in the future. The `Agent` shall stop calling the **read** operation once either ***max_total_samples*** have been returned, or else ***max_total_elapsed_time*** has elapsed.

The member ***max_total_samples*** may take the special value MAX_ELAPSED_TIME_UNLIMITED. This value shall indicate that there is no termination condition based on the elapsed time.

The member ***min_pace_period*** may take the special value MIN_PACE_PERIOD_NONE. This value shall indicate that there is no minimum time interval between samples.

8.3.5.10 DATA

The **DATA** submessage shall be sent by the XRCE `Agent` to the XRCE Client in response to a **READ_DATA** message when the **read** operation performed by the XRCE `Agent` returns STATUS_OK. If the **read** operation returns any other status the XRCE `Agent` shall send a **STATUS** message, not a **DATA** message.

The submessage contains output parameters of the **read** operation on the XRCE `DataReader` that was triggered by the **READ_DATA** message. The ***returnStatus*** is implied to be STATUS_OK.

A single **READ_DATA** message may result on multiple, possible an open-ended sequence, of **DATA** submessages sent as a response by the XRCE Agent. The **DATA** messages will continue to be sent until the one of the terminating conditions on the **READ_DATA** operation is reached, or until it is explicitly cancelled.

The ***request_id*** and ***object_id*** within the **DATA** payload shall match the namesake fields in the corresponding **READ_DATA** message.

8.3.5.10.1 flags

The **DATA** submessage uses the lowest order 4 bits of the ***flags***. The ***flags*** shall be interpreted the same way as the flags of the WRITE_DATA submessage. See 8.3.5.8.1.

8.3.5.10.2 payload

The format the ***payload*** shall match the one requested in the **READ_DATA** message having the matching ***request_id***. It shall also match the `DataFormat` encoded in the flags as shown in Table 13 – Flag bits used by the WRITE_DATA and DATA submessages. The correspondence shall be as shown in Table 15 below.

Table 15 – Payload format associated with each DataFormat

DataFormat	***Contents of payload.*** ***See Annex A*** IDL Types ***for the definition***
FORMAT_DATA	struct DATA_Payload_Data
FORMAT_SAMPLE	struct DATA_Payload_Sample
FORMAT_DATA_SEQ	struct DATA_Payload_DataSeq
FORMAT_SAMPLE_SEQ	struct DATA_Payload_SampleSeq
FORMAT_PACKED_SAMPLES	struct DATA_Payload_PackedSamples

The types referenced in Table 15 shall be as defined in Annex A IDL Types:

```
@extensibility(FINAL)
struct SampleData {
    XCDRSerializedBuffer serialized_data;
};
@extensibility(FINAL)
struct Sample {
    SampleInfo   info;
    SampleData   data;
};

@extensibility(FINAL)
struct DATA_Payload_Data : RelatedObjectRequest {
    SampleData              data;
};
```

```
@extensibility(FINAL)
struct DATA_Payload_Sample : RelatedObjectRequest {
      Sample                sample;

@extensibility(FINAL)
struct DATA_Payload_DataSeq : RelatedObjectRequest {
    sequence<SampleData>    data_seq;
};

@extensibility(FINAL)
struct DATA_Payload_SampleSeq : RelatedObjectRequest {
    sequence<Sample>        sample_seq;
};

@extensibility(FINAL)
struct DATA_Payload_PackedSamples : RelatedObjectRequest {
    PackedSamples           packed_samples;
};
```

All the **DATA** payload representations extend `RelatedObjectRequest`. The ***request_id*** and ***object_id*** within the `RelatedObjectRequest` shall match the namesake fields in the corresponding **READ_DATA** message

8.3.5.11 ACKNACK

The **ACKNACK** submessage is used to enable a transport independent reliability protocol to be implemented. If the transport used for a session is able to reliably send messages in case of disconnection or a wakeup/sleep cycle then these messages may not be required.

This specification does not dictate whether **ACKNACK** messages shall be sent only in response to **HEARTBEAT**. messages or can also be sent whenever one side detects message loss. However, in general it is expected that it is the XRCE `Client` that initiates any synchronization and therefore the XRCE `Agent` will only send **ACKNACK** messages in response to **HEARTBEAT** messages. This is because a XRCE `Client` may not be continually available as it goes on sleep cycles.

The **ACKNACK** message is directed to the same session and stream indicated in the `MessageHeader` (see 8.3.2). For this reason, it does not contain an `ObjectId`.

The ***sequenceNr*** present in the `MessageHeader` (see 8.3.2) shall not be interpreted as a sequence number belonging to the session. Rather it is interpreted as an epoch that mat be used to discard old or duplicate **ACKNACK** messages.

8.3.5.11.1 flags

The **ACKNACK** submessage does not define any additional flag bits beyond the common ones specified in 8.3.4.2.

8.3.5.11.2 payload

The **ACKNACK** submessage payload shall contain information about the state of the `Session` and `Stream`. The payload shall contain the XCDR representation of the `ACKNACK_Payload` object defined in Annex A IDL Types:

```
struct ACKNACK_Payload {
    short  first_unacked_seq_num;
    octet[2] nack_bitmap;
};
```

The ***first_unacked_seq_num*** shall indicate that all sequence numbers up to but not including it have been received.

The ***nack_bitmap*** shall indicate missing sequence numbers, starting from ***first_unacked_seq_num***.

For example, an `ACKNACK_Payload` having ***first_unacked_seq_num*** set to 100 and ***nack_bitmap*** set to 0x4009 (in binary 0100 0000 0000 1001) would indicate that all sequence numbers up to and including 99 have been received. Furthermore it would also indicate that sequence numbers 100, 103, and 114 are missing.

8.3.5.12 HEARTBEAT

The **HEARTBEAT** submessage is used to enable a transport independent reliability protocol to be implemented.

This specification does not limit a session to use a particular type of transport. If a session transport is able to reliably send messages in case of disconnection or a wakeup/sleep cycle then these messages may not be required.

This specification does not dictate the timing of **HEARTBEAT** messages. However, in general it is expected that it is the XRCE `Agent` will only send **HEARTBEAT** messages when it has some indication that the XRCE `Client` is active and not in a sleep cycle. This is to avoid awakening the XRCE `Client` unnecessarily.

The **HEARTBEAT** message is directed to the same `Session` and `Stream` indicated in the `MessageHeader`. For this reason it does not contain an `ObjectId`.

The ***sequenceNr*** present in the `MessageHeader` (see 8.3.2) shall not be interpreted as a sequence number belonging to the session. Rather it is interpreted as an epoch that mat be used to discard old or duplicate **ACKNACK** messages.

8.3.5.12.1 flags

The **HEARTBEAT** submessage does not define any additional flag bits beyond the common ones specified in 8.3.4.2.

8.3.5.12.2 payload

The **HEARTBEAT** submessage payload shall contain information about the state of the `Session` and `Stream`. The payload shall contain the XCDR representation of the `HEARTBEAT_Payload` object defined in Annex A IDL Types:

```
@extensibility(FINAL)
struct HEARTBEAT_Payload {
    short  first_unacked_seq_nr;
    short  last_unacked_seq_nr;
};
```

The ***first_unacked_seq_nr*** indicates the first available message sequence number on the sending side.

The ***last_unacked_seq_nr*** indicates the first available message sequence number on the sending side.

8.3.5.13 RESET

The **RESET** submessage shall be used to reset and re-establish a session. It contains no payload. It shall cause the XRCE Agent to reset all state associated with the ***session_id*** indicated in the submessage header.

8.3.5.13.1 flags

The **RESET** submessage does not define any additional flag bits beyond the common ones specified in 8.3.4.2.

8.3.5.13.2 payload

The **RESET** submessage shall have an empty payload.

8.3.5.14 FRAGMENT

This **FRAGMENT** submessage is used to enable sending of other submessages whose length exceeds the transport MTU.

The **FRAGMENT** message shall only be sent within reliable streams.

When a message is broken into fragments all **FRAGMENT** submessage except for the last shall have the 'Last Fragment' bit in the flags set to 0. The last **FRAGMENT** submessage shall have the 'Last Fragment' flag set 1.

Upon reception of the last fragment submessage the `Agent` shall concatenate the payload bytes of all **FRAGMENT** messages for that `Stream` in the order of the stream sequence number without sequence number gaps. The concatenated payloads shall be interpreted as XRCE submessages as if they had been received following the **HEADER** that came with the last fragment.

8.3.5.14.1 flags

The **FRAGMENT** submessage uses the lowest order 2 bits of the ***flags***:

- Bit 0 indicates the 'Endianness' as specified in 8.3.4.2.
- Bit 1, the 'Last Fragment' bit, indicates the last fragment in the sequence.

8.3.5.14.2 payload

The payload of the **FRAGMENT** submessage is opaque. The `Agent` shall cache the payload bytes of all **FRAGMENT** submessages for a `Stream` in the order of the stream sequence number until the last **FRAGMENT** submessage is received.

8.4 Interaction Model

8.4.1 General

This section describes typical message flows.

The XRCE protocol is defined such that it is possible to implement clients that minimize discovery and setup traffic. For this reason some of the message flows are optional and may be replaced by out-of-band configuration of the XRCE `Client` and `Agent`.

8.4.2 Sending data using a pre-configured DataWriter

The message flow below illustrates the complete set of messages used by an XRCE `Client` to write data using the XRCE `Agent`. The XRCE `Agent` has been pre-configured to create a XRCE `Application` containing a `DomainParticipant`, `Publisher` and `DataWriter`. The `DataWriter` pre-configured ***object_id*** is known to

the XRCE `Client`.

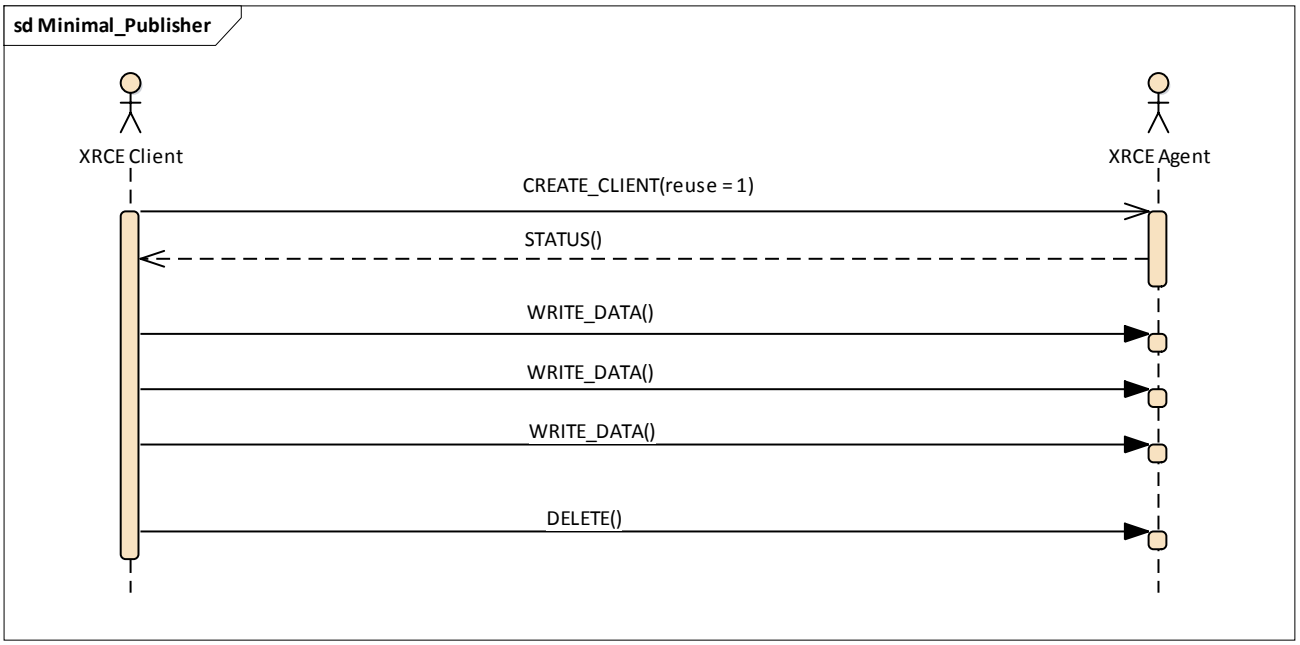


Figure 8— Message flow to send data using a pre-configured DataWriter

An XRCE Agent has been pre-configured for a Client (identified by the ClientKey) such that it recognizes the ***application_object_id*** *present in the CREATE_CLIENT message. The reception of the CREATE_CLIENT triggers the creation or reuse of the corresponding XRCE objects. These include XRCE DataWriters with their corresponding DDS DataWriters. Subsequent WRITE_DATA messages reference the ObjectId of those DataWriters in order to publish data using DDS.*

8.4.3 Receiving data using a pre-configured DataReader

The message flow below illustrates the complete set of messages used by an XRCE `Client` to receive data via the XRCE `Agent`. The XRCE `Agent` has been pre-configured to create a XRCE `Application` containing a `DomainParticipant`, `Subscriber` and `DataReader`. The `DataReader` pre-configured ***object_id*** is known to the XRCE `Client`.

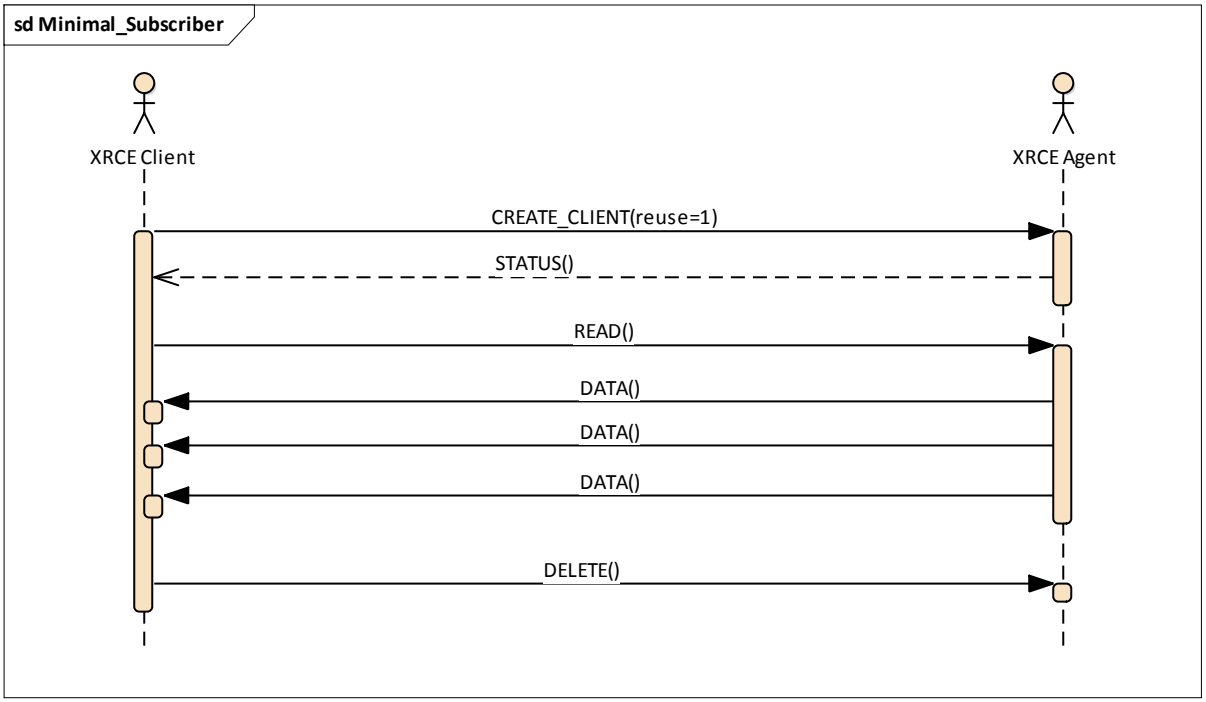


Figure 9— Message flow to receive data using a pre-configured DataReader

*An Agent has been pre-configured for a Client (identified by the ClientKey) such that it recognizes the **application_object_id** present in the CREATE_CLIENT message. The reception of the CREATE_CLIENT triggers the creation or reuse of the corresponding XRCE objects. These include XRCE DataReaders with their corresponding DDS DataReaders. A subsequent READ message references the ObjectId of those DataReaders in order to receive data from the DDS domain.*

8.4.4 Discovering an Agent

The message flow below illustrates the messages needed for an XRCE `Client` to discover XRCE `Agents`. This flow is only required when the `Client` is not pre-configured with the `TransportLocator` of the XRCE `Agent`. It allows an XRCE `Client` to be configured to content one or more `TransportLocators` (which may include multicast addresses) in order to dynamically discover the presence and actual Address of the `Agents`.

As a result of this process, the XRCE `Client` may discover more than one XRCE `Agent`. In that case it may use the information received about the XRCE `Agent` configuration (e.g. the fields ***version***, ***vendor_id***, or ***properties*** found within the `AGENT_Representation`) and the XRCE `Agent` activity (e.g. the ***availability*** field within the `ActivityInfo`) to select the most appropriate XRCE `Agent` and even connect to more than one XRCE `Agents`.

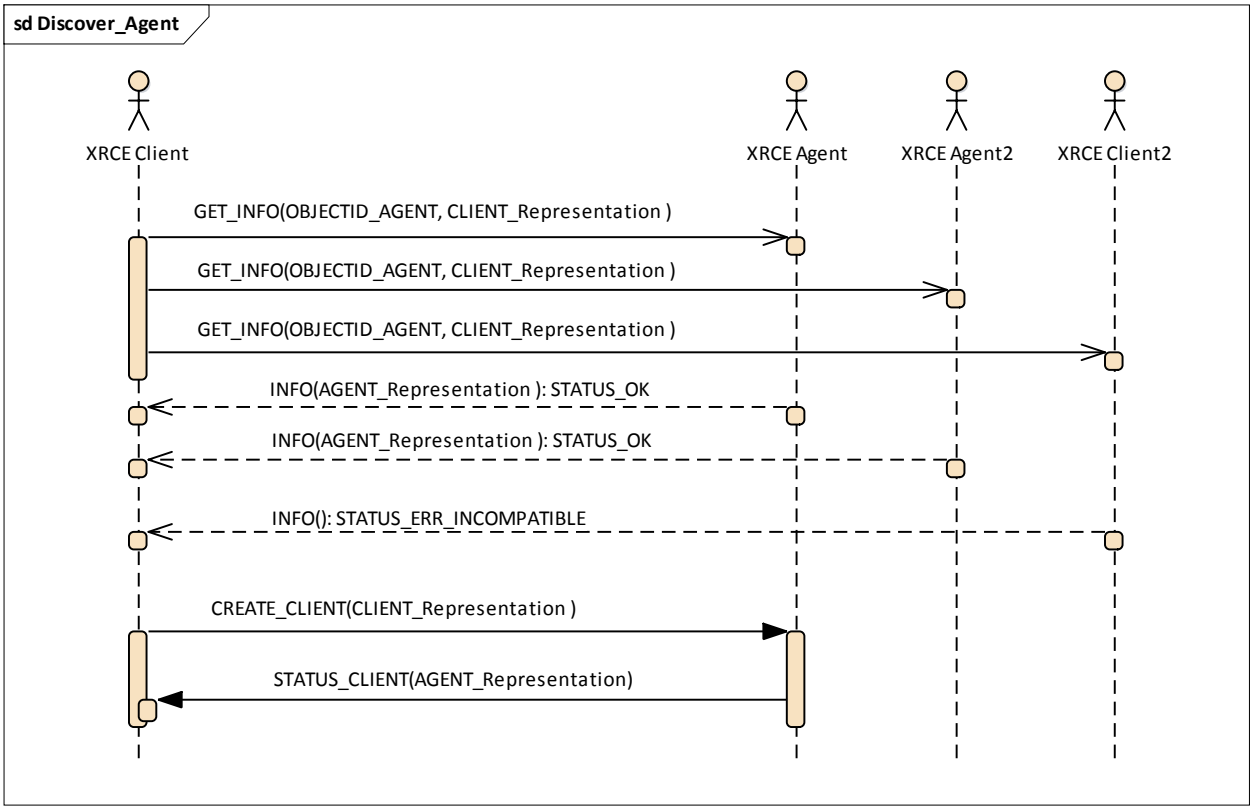


Figure 10— Message flow for a Client to connect to an Agent

An XRCE Client connects to an Agent using the CREATE_CLIENT message. The Agent responds with a STATUS_AGENT indicating whether the connection succeeded and the ClientProxy was created on behalf of the XRCE Client.

8.4.5 Connecting to an Agent

The message flow below illustrates the messages needed for an XRCE `Client` to connect to XRCE `Agent`. After the `Client` is connected it may create resources or invoke operations on existing resources.

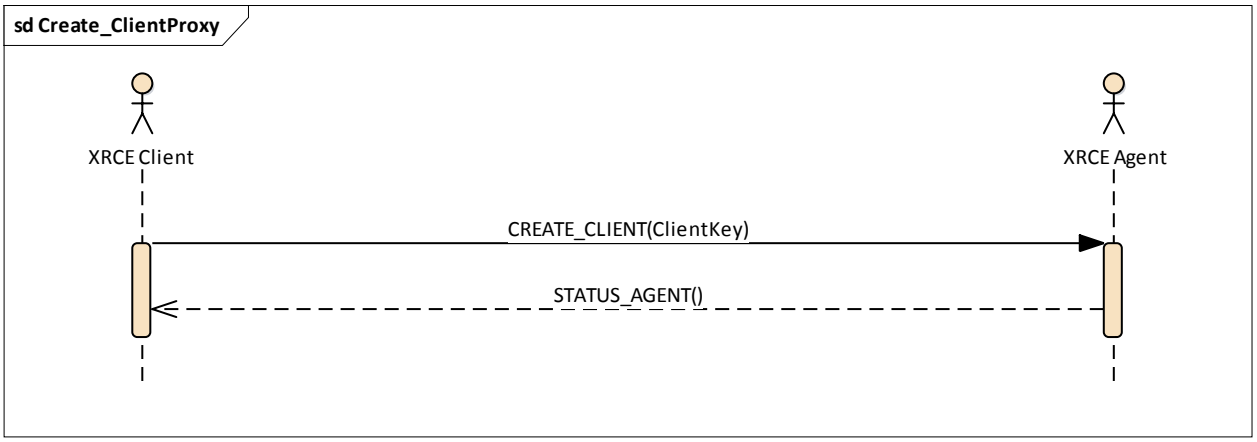


Figure 11— Message flow for a Client to connect to an Agent

An XRCE Client connects to an Agent using the CREATE_CLIENT message. The Agent responds with a STATUS_AGENT indicating whether the connection succeeded and the ClientProxy was created on behalf of the XRCE Client.

8.4.6 Creating a complete Application

The message flow below illustrates the messages needed for an already connected XRCE `Client` to create a complete XRCE `Application`.

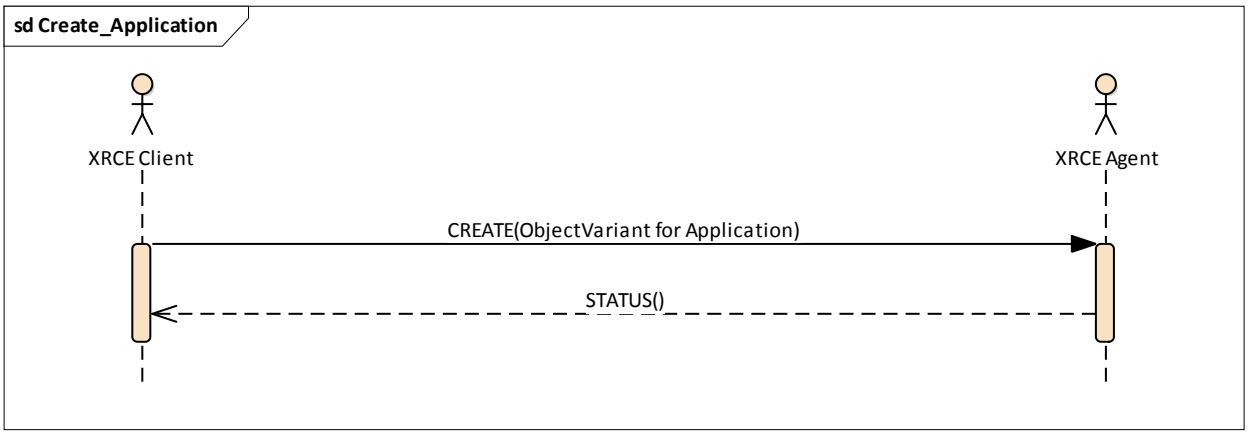


Figure 12— Message flow for a Client to create an Application

An XRCE Client uses the CREATE message to create an XRCE Application. The CREATE message carries a CREATE_Payload containing an ObjectVariant with ObjectKind set to OBJK_APPLICATION. The corresponding OBJK_APPLICATION_Representation may use the REPRESENTATION_BY_REFERENCE to refer to an Application pre-configured in the Agent or it may use the REPRESENTATION_AS_XML_STRING to fully describe the Application including any necessary Types, Qos, and DDS Entities.

8.4.7 Defining Qos configurations

The message flow below illustrates the messages needed for an already connected XRCE `Client` to dynamically define XRCE `QosProfiles` which may later be used to create other XRCE `Objects`.

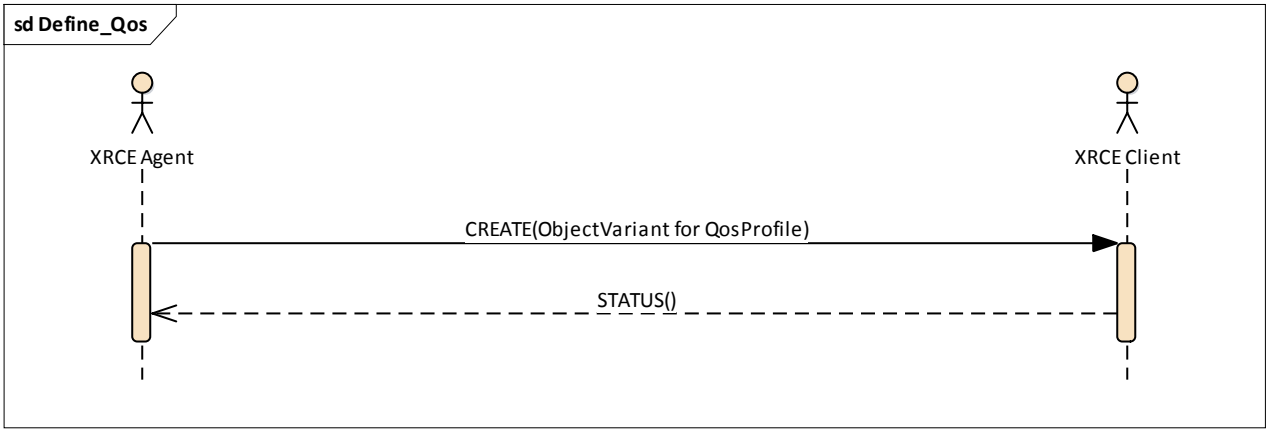


Figure 13— Message flow for a Client to define Qos Profiles

An XRCE Client uses the CREATE message to define Qos Profile. The CREATE message carries a CREATE_Payload containing an ObjectVariant with ObjectKind set to OBJK_QOSPROFILE. The corresponding OBJK_QOSPROFILE_Representation may use the REPRESENTATION_AS_XML_STRING to fully describe the Qos Profile.

8.4.8 Defining Types

The message flow below illustrates the messages needed for an already connected XRCE `Client` to dynamically define XRCE `Types` which may later be used to create XRCE `Topic` objects.

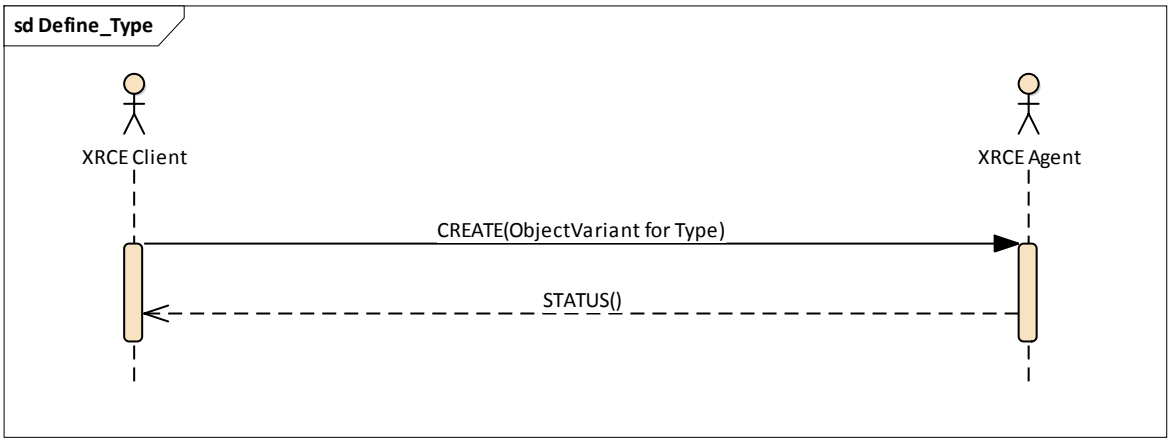


Figure 14— Message flow for a Client to define Types

An XRCE Client uses the CREATE message to create an XRCE Type. The CREATE message carries a CREATE_Payload containing an ObjectVariant with ObjectKind set to OBJK_TYPE. The corresponding OBJK_TYPE_Representation may use the REPRESENTATION_AS_XML_STRING to fully describe the DDS-XTYPES Type including any referenced types.

8.4.9 Creating a Topic

The message flow below illustrates the messages needed for an already connected XRCE `Client` to dynamically create a XRCE `Topic`, which may later be used to create XRCE `DataWriter` and `DataReader` objects.

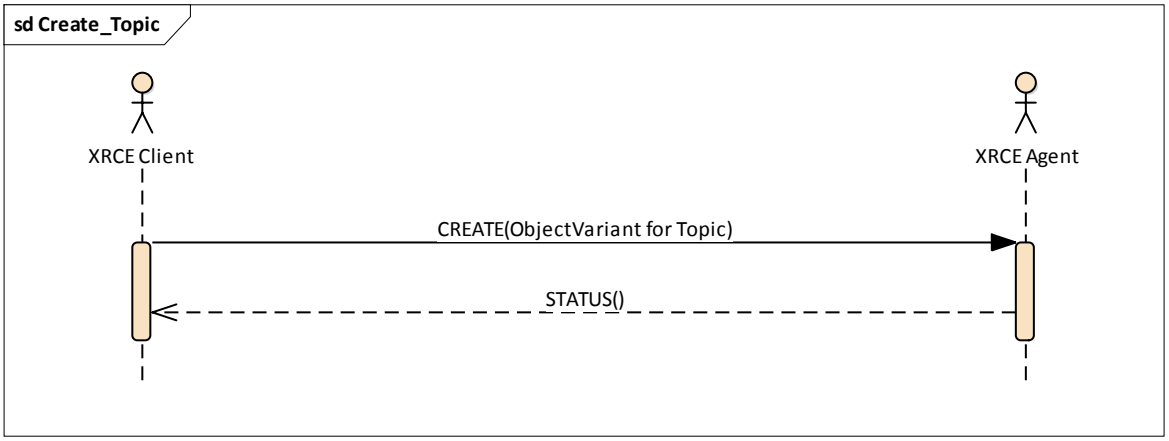


Figure 15— Message flow for a Client to define a Topic

An XRCE Client uses the CREATE message to create an XRCE Topic. The CREATE message carries a CREATE_Payload containing an ObjectVariant with ObjectKind set to OBJK_TOPIC. The corresponding OBJK_TOPIC_Representation may use the REPRESENTATION_IN_BINARY or the REPRESENTATION_AS_XML_STRING to fully define the Topic.

8.4.10 Creating a DataWriter

The message flow below illustrates the messages needed for an already connected XRCE `Client` to dynamically create a XRCE `DataWriter` with all the resources needed resources to publish data.

The XRCE `Agent` may have a-priory knowledge of QoS profiles, allowing the XRCE `Client` to refer to those by name rather than explicitly define them. Alternatively the XRCE `Client` may include them as part definition of the XRCE `DataWriter` resource.

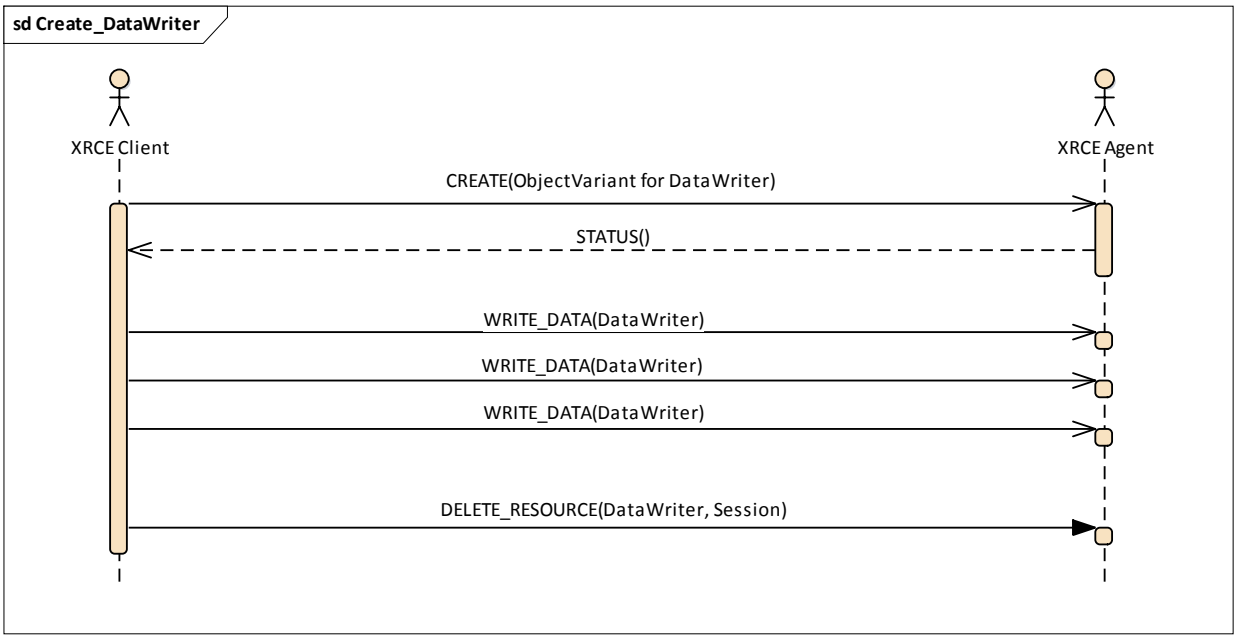


Figure 16— Message flow for a Client to create a DataWriter

An XRCE Client uses the CREATE message to create an XRCE DataWriter. The CREATE message carries a CREATE_Payload containing an ObjectVariant with ObjectKind set to OBJK_DATAWRITER. The corresponding DATAREADER_Representation may use the REPRESENTATION_IN_BINARY or the REPRESENTATION_AS_XML_STRING to fully define the DataWriter. Both these representations allow specification of the DataWriter Qos. The DATAREADER_Representation may also use the REPRESENTATION_BY_REFERNCE to refer to a DataWriter definition known to the Agent.

8.4.11 Creating a DataReader

The message flow below illustrates the messages needed for an already connected XRCE `Client` to dynamically create a XRCE `DataReader` with all the resources needed resources to publish data.

The XRCE `Agent` may have a-priory knowledge of QoS profiles, allowing the XRCE `Client` to refer to those by name rather than explicitly define them. Alternatively the XRCE `Client` may include them as part definition of the XRCE `DataReader` resource.

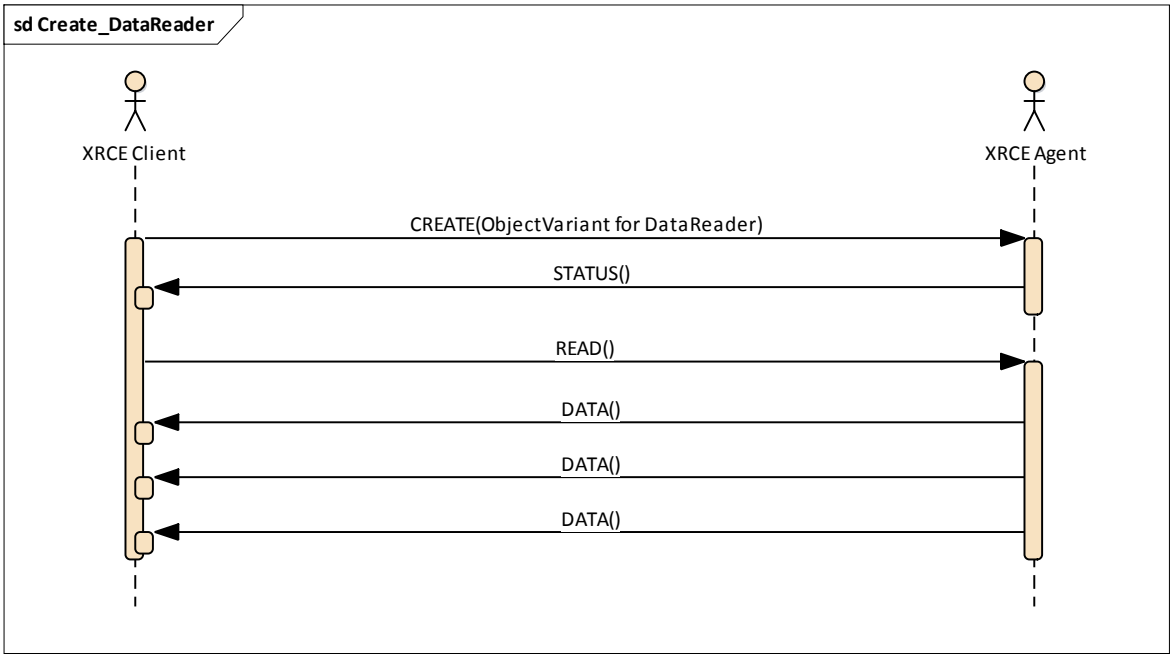


Figure 17— Message flow for a Client to create a DataReader

An XRCE Client uses the CREATE message to create an XRCE DataReader. The CREATE message carries a CREATE_Payload containing an ObjectVariant with ObjectKind set to OBJK_DATAREADER. The corresponding OBJK_DATAREADER_Representation may use the REPRESENTATION_IN_BINARY or the REPRESENTATION_AS_XML_STRING to fully define the DataReader. Both these representations allow specification of the DataReader Qos. The OBJK_DATAREADER_Representation may also use the REPRESENTATION_BY_REFERENCE to refer to a DataReader definition known to the Agent.

8.4.12 Getting Information on a Resource

The message flow below illustrates how an XRCE `Client` may query information on a resource. An XRCE `Client` may use this mechanism to determine the QoS of any of the DDS proxy entities that the XRCE `Agent` manages on behalf of the XRCE `Client`. It may also be used to read QoS profiles and type declarations that are known to the XRCE `Agent`.

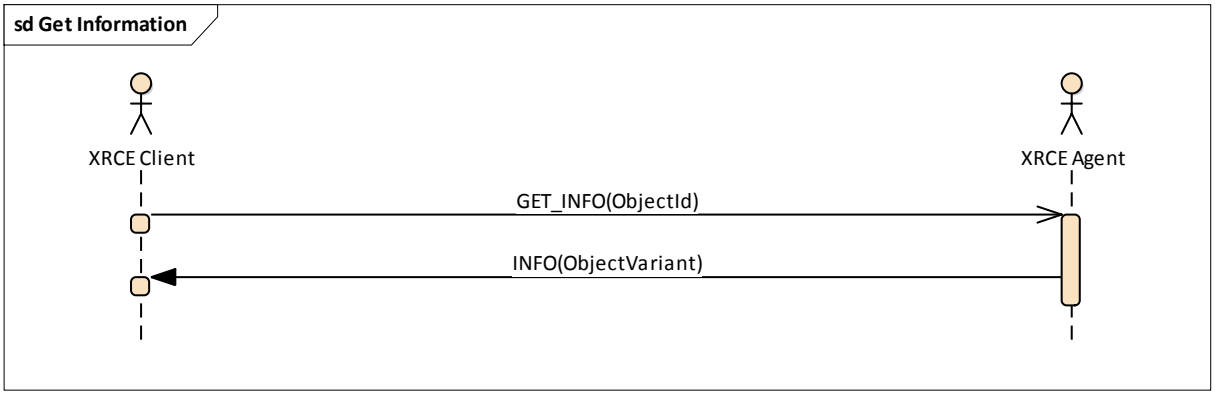


Figure 18— Message flow for a Client to create a DataReader

An XRCE Client uses the GET_INFO message to get information from an XRCE Object identified by its ObjectId. The XRCE Agent responds with an INFO message containing an ObjectVariant. The ObjectKind of the ObjectVariant is the appropriate for the specified ObjectId.

8.4.13 Updating a Resource

The message flow below illustrates how a XRCE `Client` may update an XRCE `DataReader`. A XRCE `Client` may use this mechanism to change the QoS parameters of any of the DDS proxy entities that the XRCE `Agent` manages on behalf of the XRCE `Client`.

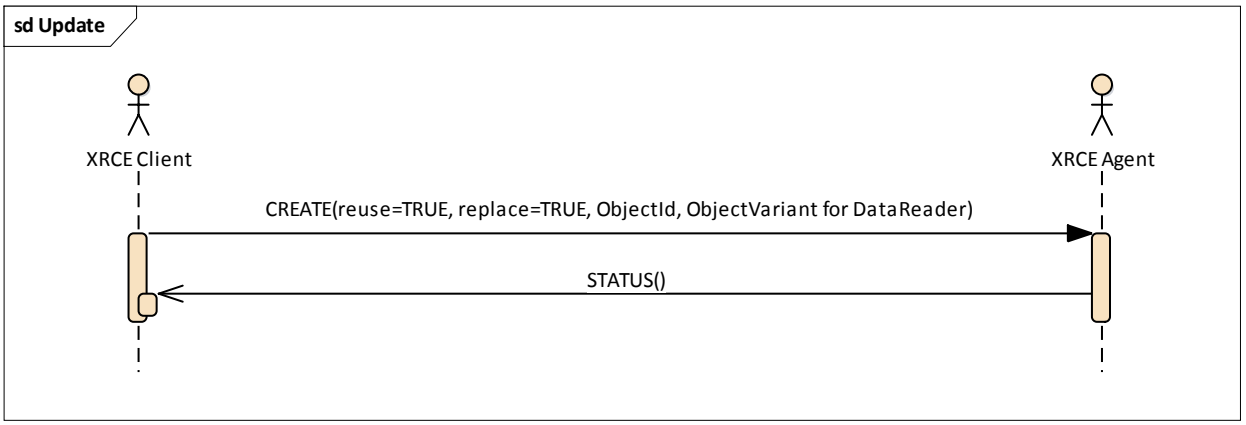


Figure 19— Message flow for a Client to create a DataReader

*An XRCE Client uses the CREATE message with the attribute **reuse** set to TRUE and the attribute **replace** set to TRUE to indicate it wants to update the Object identified by the ObjectId. The CREATE message contains an ObjectVariant with ObjectKind set to the appropriate value for the specified ObjectId. The XRCE Agent updates the Object using the new configuration contained in the ObjectVariant and responds with a STATUS message.*

8.4.14 Reliable Communication

Reliability is implemented separately for each `Stream`, and only for the reliable streams which are identified by the ***stream_id*** value being between 0x80 and 0xFF. See clause 8.3.2.2 Streams and the streamId.

A `Stream` has exactly two endpoints, the sending endpoint and the receiving endpoint. Note that for some streams the sender is the XRCE `Client`, e.g. when the XRCE `Client` uses a stream to write data to the XRCE `Agent`. Likewise in other streams the sender may be the XRCE `Agent`, for example when the XRCE `Agent` uses a stream to send the data the XRCE `Client` requested in a **READ** operation.

The sender and receiver endpoint on a `Stream` each execute its own protocol state machine. These are illustrated in the following subsections.

Sequence number arithmetic and comparisons shall use Serial Number Arithmetic as specified in clause 8.3.2.3 sequenceNr.

8.4.14.1 Reliable sender state machine

The protocol executed by the endpoint that is sending on a stream is shown in Figure 20

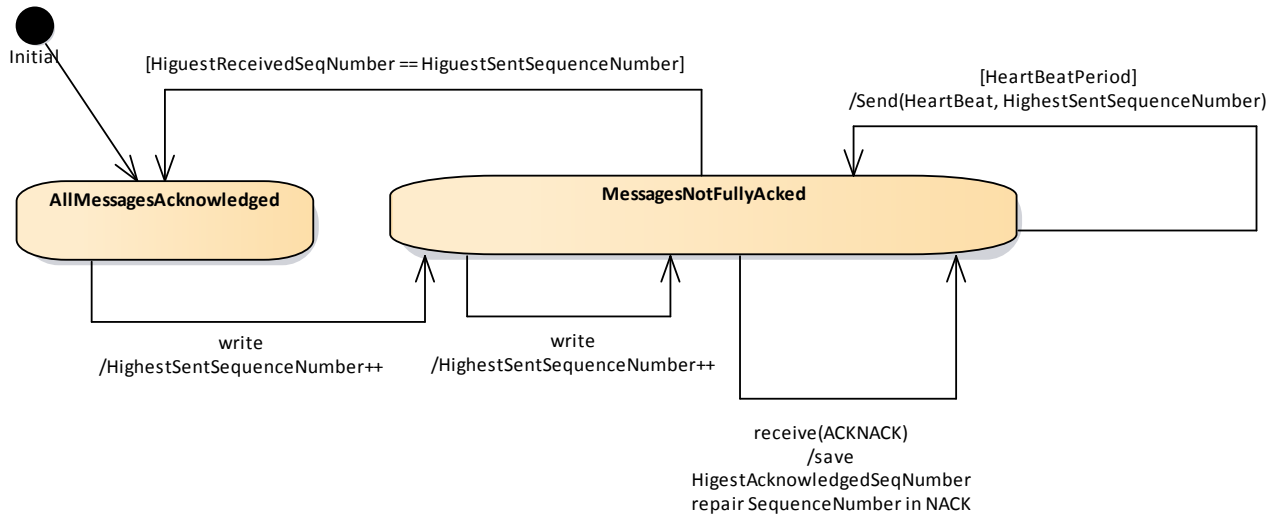


Figure 20— Reliable protocol state-machine for the sender on a stream

The sender maintains two state variables associated with the stream. The ***HighestSentSequenecNumber*** and the ***HighestAcknowledgedSequenceNumber***.

Each time a message is sent the HighestSentSequenecNumber is increased. The reception of **ACKNACK** messages updates the HiguestAcknowledgedSequenceNumber.

While the ***HighestAcknowledgedSequenceNumber*** is less than the ***HighestSentSequenceNumber*** the sender sends HeartBeat messages that announce the ***HighestSentSequenecNumber*** to the receiver. These HeartBeat messages may be periodic or optimized using on vendor specific mechanism. The requirement is that they are sent at some rate until ***HighestAcknowledgedSequenceNumber*** matches the ***HighestSentSequenceNumber***.

8.4.14.2 Reliable receiver state machine

The protocol executed by the endpoint that is receiving on a reliable stream is shown in Figure 21

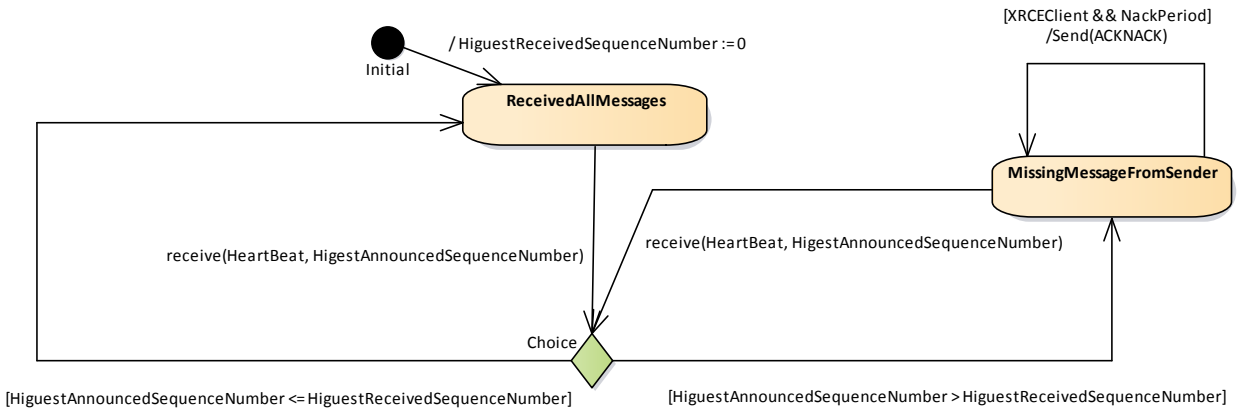


Figure 21— Reliable protocol state-machine for the receiver on a stream

The receiver maintains two state variables associated with the stream. The ***HighestReceivedSequenceNumber*** and the ***HighestAnnouncedSequenceNumber***.

Each time a Message is received the ***HighestReceivedSequenceNumber*** may be updated (assuming all previous messages have been received). The ***HighestAnnouncedSequenceNumber*** may also be adjusted.

Each time a **HEARTBEAT** is received the ***HighestAnnouncedSequenceNumber*** may be adjusted.

If the receiver is a XRCE `Client`, then while the ***HiguestReceivedSequenceNumber*** is less than the ***HighestAnnouncedSequenceNumber***, the received sends **ACKNACK** messages to request the messages corresponding to the missing sequence numbers. These **ACKNACK** messages may be periodic or optimized using on vendor specific mechanism.

If the receiver is the XRCE `Agent`, then it only sends **ACKNACK** messages in response to receiving a **HEARTBEAT**. This is done to avoid overwhelming the XRCE `Client` or waking it up at a non-opportune time.

8.5 XRCE Object Operation Traceability

This clause summarizes the messages used to implement each operation on the XRCE `Object` model ensuring that all operations have been covered.

The messages used trigger each operation and receive the result are summarized in Table 16

Table 16 – Predefined XRCE Objects from parsing the Example XML configuration XML file

XRCE Object Kind	Operation	Message used for Invocation	Message used for Return
XRCE Root	create_client	CREATE_CLIENT	STATUS_AGENT
XRCE Root	get_info	GET_INFO	INFO
XRCE Root	delete_client	DELETE	STATUS_AGENT
XRCE ProxyClient	create	CREATE (flags for creation)	STATUS
XRCE ProxyClient	update	CREATE (flags for reuse)	STATUS
XRCE ProxyClient	get_info	GET_INFO	INFO
XRCE ProxyClient	delete	DELETE	STATUS
XRCE DataWriter	write	WRITE_DATA, FRAGMENT	STATUS
XRCE DataReader	read	READ_DATA	DATA, FRAGMENT, STATUS

9 XRCE Agent Configuration

9.1 General

The XRCE `Agent` may be configured such that it has a priori knowledge XRCE `Objects`. This allows XRCE `Clients` to reference and create XRCE `Objects` in a very compact manner using the representation format REPRESENTATION_BY_REFERENCE, see clause 7.7.3.3.1 REPRESENTATION_BY_REFERENCE format.

This specification provides two standard mechanisms to configure the XRCE `Agent`. Implementations may also provide additional mechanisms:

- Remote configuration using the XRCE Protocol
- Local file-based configuration

These mechanisms are described in the clauses that follow.

9.2 Remote configuration using the XRCE Protocol

An application may use a XRCE `Client` with the only purpose of defining and creating XRCE `Objects` that are intended for other applications. This type of application is called a XRCE `ConfigurationClient`.

The protocol used by the XRCE `ConfigurationClient` is the same used by any other XRCE `Client`. The only difference is that an XRCE `ConfigurationClient` never uses the READ_DATA or WRITE messages. It only uses the messages that create, update, or retrieve information about the XRCE objects.

Any other XRCE `Client` can reference XRCE `Objects` created by an XRCE `ConfigurationClient`.

A typical use of the remote configuration mechanism are tools that may be used to configure an `Agent` prior to deployment or to interactively configure the system.

Note that the XRCE `ConfigurationClient` may be communicating with the `Agent` using a different network or transport, which may not have the same constraints as a typical XRCE `Client`.

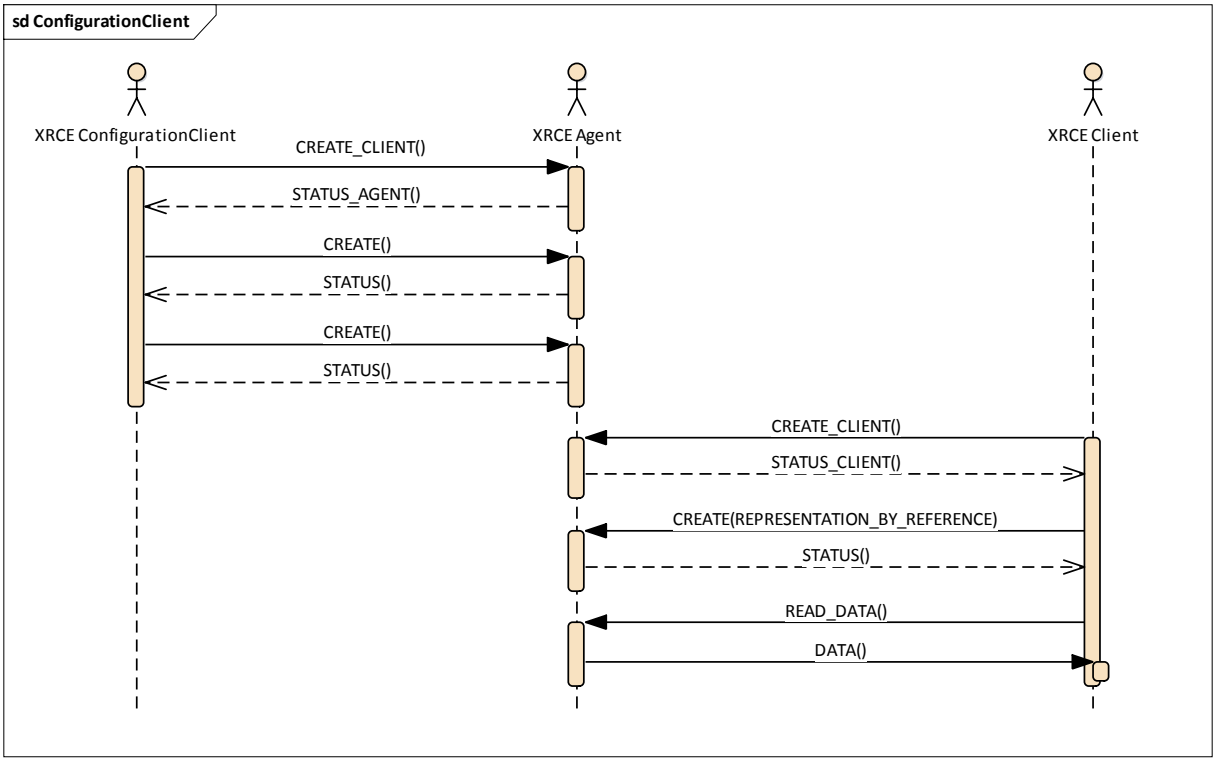


Figure 22— Message flow for a ConfigurationClient

An XRCE ConfiguratioClient uses CREATE messages with representation formats REPRESENTATION_IN_BINARY or REPRESENTATION_AS_XML_STRING to define and create XRCE Objects in the XRCE Agent. These XRCE Objects are later referenced by a different XRCE Client using the representation formats REPRESENTATION_BY_REFERENCE.

9.3 File-based Configuration

The XRCE `Agent` shall provide a configuration or run-time option to load an XML file formatted according to the schema defined in the [DDS-XML] machine-readable file **dds-xml_system_example.xsd**.

The XRCE `Agent` shall parse the XML file and for each of the elements defined in Table 17, it shall construct the corresponding XRCE `Object` specified in Table 17. All the created XRCE `Objects` shall be made available to XRCE clients such that they may refer to them using the representation format `REPRESENTATION_BY_REFERENCE`.

Table 17 – XRCE Object created from the elements in the configuration XML file

XML Element(s)	XRCE Object	REPRESENTATION_BY_REFERENCE
<types>	XRCE Type.	The created XRCE Types shall be referenceable using their fully qualified name, which includes the names of enclosing modules. For example: “MyModule::MyNestedModule::MyStructType”
<qos_profile> **(Child of <qos_library>)**	XRCE QosProfile.	The created XRCE Types shall be referenceable using their fully qualified name, which includes the names of enclosing Qos Profile Library. For example: “MyProfileLibrary::MyQosProfile”

<domain> **(Child of <domain_library>)**	XRCE Domain.	The created XRCE Domain shall be referenceable using their fully qualified name, which includes the names of enclosing Domain Library. For example: "MyDomainLibrary::MyDomain"
<topic> **(Child of <domain>)**	XRCE Topic	The created XRCE Topic shall be referenceable using its name from any DomainParticipant that references the Domain where the Topic is defined. For example: "ExampleTopic"
<application> **(Child of <application_library>)**	XRCE Application.	The created XRCE Application shall be referenceable using their fully qualified name, which includes the names of enclosing Application Library. For example: "MyApplicationLibrary::MyApplication"
<domain_participant> **(Child of** **<domain_participant _library>)**	XRCE DomainParticipant	The created XRCE DomainParticipant shall be referenceable using their fully qualified name, which includes the names of enclosing DomainParticipant Library. For example: "MyParticipantLibrary::MyParticipant"
<topic> **(Child of <domain_participant>)**	XRCE Topic	The created XRCE Topic shall be referenceable using its name from any objects in the same DomainParticipant. For example: "ExampleTopic"
<publisher> <subscriber> **(Child of** **<domain_participant>)**	XRCE Publisher XRCE Subscriber	The created XRCE Publisher or Subscriber shall be referenceable using their name. No qualification is necessary since these entities are always referenced within the scope of a DomainParticipant. For example: "MyPublisher", "MySubscriber"
<data_writer> < data_reader> **(Child of** **<domain_participant>)**	XRCE DataWriter XRCE DataReader	The created XRCE DataWriter or DataReader shall be referenceable using their name. No qualification is necessary since these entities are always referenced within the scope of a Publisher or Subscriber. For example: "MyWriter", "MyReader"

The XRCE `Objects` created from the file-based configuration shall have their `ObjectId` automatically derived from the `REPRESENTATION_BY_REFERENCE` string. Specifically, the `ObjectIdPrefix` (see 7.7.6) shall be set to the first 2 bytes of the MD5 hash computed on the `REPRESENTATION_BY_REFERENCE` string. The MD5 treats each string character as a byte and does not include the NUL terminating character of the string.

For example assuming the REPRESENTATION_BY_REFERENCE string is "MyWriter" in that case:

- The MD5 hash shall be: 0x03e26181adfef529038bf0dce7cab871
- The `ObjectIdPrefix` shall be the two-byte array: {0x03, 0xe2}.
- The `ObjectIdPrefix` shall be computed by combining the `ObjectIdPrefix` with the `ObjectKind` as specified in clause 7.7.6.

9.3.1 Example Configuration File

The following XML file could be used to configure a XRCE Agent.

```
<?xml version="1.0" encoding="UTF-8"?>

<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns="http://www.omg.org/dds"
     xsi:schemaLocation="http://www.omg.org/spec/DDS-XML/20170301/dds-
xml_system_example.xsd">

    <types>
        <module name="ShapesDemoTypes" >
            <const name="MAX_COLOR_LEN" type="int32" value="128" />
            <struct name="ShapeType">
                <member name="color" key="true" type="string"
                        stringMaxLength="MAX_COLOR_LEN" />
                <member name="x" type="int32" />
                <member name="y" type="int32" />
                <member name="shapesize" type="int32" />
                    </struct>
        </module>
    </types>

    <qos_library name="MyQosLibrary">
        <qos_profile name="MyQosProfile">
            <datareader_qos>
                <durability>
                    <kind>TRANSIENT_LOCAL_DURABILITY_QOS</kind>
                </durability>
                <reliability>
                    <kind>RELIABLE_RELIABILITY_QOS</kind>
                </reliability>
                <history>
                   <kind>KEEP_LAST_HISTORY_QOS</kind>
                    <depth>6</depth>
                </history>
            </datareader_qos>

            <datawriter_qos>
                <durability>
                    <kind>TRANSIENT_LOCAL_DURABILITY_QOS</kind>
                </durability>
                <reliability>
                    <kind>RELIABLE_RELIABILITY_QOS</kind>
                </reliability>
                <history>
                    <kind>KEEP_LAST_HISTORY_QOS</kind>
                    <depth>20</depth>
                </history>
                <lifespan>
                    <duration>
                        <sec>10</sec>
                        <nanosec>0</nanosec>
                    </duration>
                </lifespan>
            </datawriter_qos>
        </qos_profile>
```

```
    </qos_library>


    <application_library name="MyApplications">
        <application name="ShapesDemoApp">
            <domain_participant name="MyParticipant"
                                domain_ref="ShapesDomainLibrary::ShapesDomain">
                <register_type name="ShapeType" type_ref="ShapeType" />

                <topic name="Square" register_type_ref="ShapeType" />
                <topic name="Circle" register_type_ref="ShapeType" />
                <topic name="Triangle" register_type_ref="ShapeType" />

                <publisher name="MyPublisher">
                    <data_writer name="MySquareWriter" topic_ref="Square">
                        <datawriter_qos base_name="MyQosLibrary::MyQosProfile"/>
                    </data_writer>
                    <data_writer name="MyCircleWriter" topic_ref="Circle" />
                </publisher>

                <subscriber name="MySubscriber">
                    <data_reader name="MyTriangleRdr" topic_ref="Triangle">
                        <datareader_qos base_name="MyQosLibrary::MyQosProfile"/>
                    </data_reader>
                </subscriber>
            </domain_participant>
        </application>
    </application_library>
</dds>
```

An XRCE Agent loading the above configuration file would have the pre-defined XRCE Objects shown in Table 18.

Table 18 – Predefined XRCE Objects from parsing the Example XML configuration XML file

XRCE Object Kind	REPRESENTATION_BY_REFERENCE	ObjectPrefix	ObjectId
XRCE Type	*"ShapesDemoTypes::ShapeType"*	{0x59, 0x51}	{0x59, 0x5a}
XRCE Qos Profile	*"MyQosLibrary::MyQosProfile"*	{0x3a, 0x38}	{0x3a, 0x3b}
XRCE Application	*"MyApplications::SimpleShapesDemoApplication"*	{0x1b, 0xec}	{0x1b, 0xec}
XRCE DomainParticipant	*"MyApplications::ShapesDemoApp::MyParticipant"*	{0x56, 0xcc}	{0x56, 0xc1}
XRCE Topic	*"Square"*	{0xce, 0xb4}	{0xce, 0xb2}
XRCE Topic	*"Circle"*	{0x30, 0x95}	{0x30, 0x92}
XRCE Topic	*"Triangle"*	{0x5e, 0x55}	{0x5e, 0x52}
XRCE Publisher	*"MyPublisher"*	{0x13, 0xe3}	{0x13, 0xe3}
XRCE Subscriber	*"MySubscriber"*	{0xae, 0x0d}	{0xae, 0x04}
XRCE DataWriter	*"MySquareWriter"*	{0x1c, 0xc4}	{0x1c, 0xc5}
XRCE DataWriter	*"MyCircleWriter"*	{0xcf, 0x80}	{0xcf, 0x85}
XRCE DataReader	*"MyTriangleReader"*	{0xaf, 0x32}	{0xaf, 0x36}

10 XRCE Deployments

All the operations described in the DDS-XRCE PIM pertain to the interaction of a client application with a single DDS-XRCE Agent. The scope of all the operations is therefore limited to the interactions with that DDS-XRCE Agent. Yet client applications may interact with each other despite connecting to different DDS-XRCE Agents. These interactions would happen as a consequence of the DDS-XRCE Agents creating and performing operations on DDS DomainParticipant entities, which exchange information in accordance to the DDS specification.

10.1 XRCE Client to DDS communication

The specification defines the protocol used by an XRCE Client to communicate with a XRCE Agent that proxies for Client in the DDS Domain. The primary consequence of this is that the XRCE Client can now communicate with any DDS DomainParticipant.

The DDS DomainParticipant will discover the proxy DDS Entities that the XRCE Agent creates on behalf of the Client and with use the standard DDS-RTPS Interoperability protocol to communicate wit the Agent.

The XRCE Client will communicate with the XRCE Agent using the XRCE Protocol. Using this protocol it can direct the XRCE Agent to create new DDS entities and use these entities to read and write data on the DDS Global Data Space.

This type of deployment is shown in illustrated in Figure 23 below.

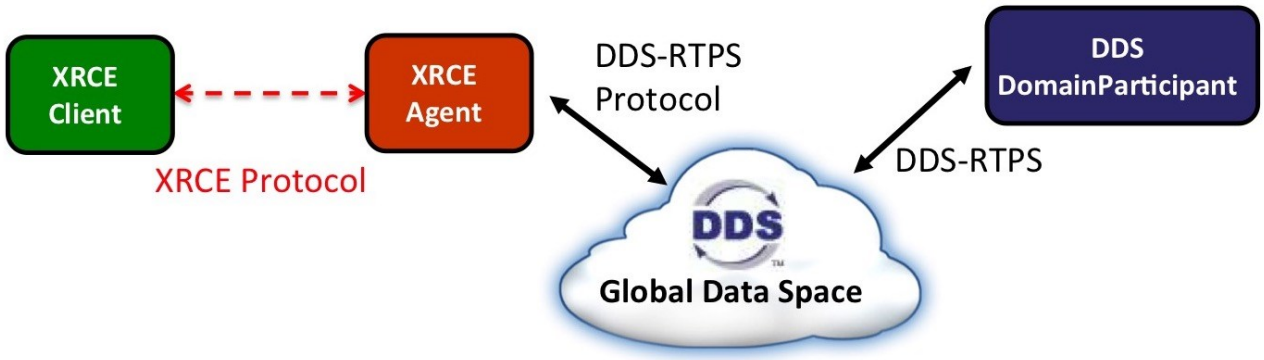


Figure 23— XRCE Agent proxying for an XRCE Client on a DDS Domain

The XRCE Client communicates with the XRCE Agent using the XRCE Protocol. The XRCE Agent communicates with other DDS DomainParticipants in the DDS Domain using the DDS-RTPS Protocol.

10.2 XRCE Client to Client via DDS

XRCE Agents appear as DDS DomainParticipants in the DDS Domain. For this reason XRCE Client applications that are connected to different XRCE Agents will communicate with each other without the need for further configuration.

Each XRCE Agent will perceive other XRCE Agents as DDS DomainParticipants, indistinguishable from any other DDS DomainParticipant and communicate with them using DDS-RTPS. The XRCE Agents will relay that communication to their respective XRCE Clients.

This type of scenario is shown in illustrated in Figure 24 below.

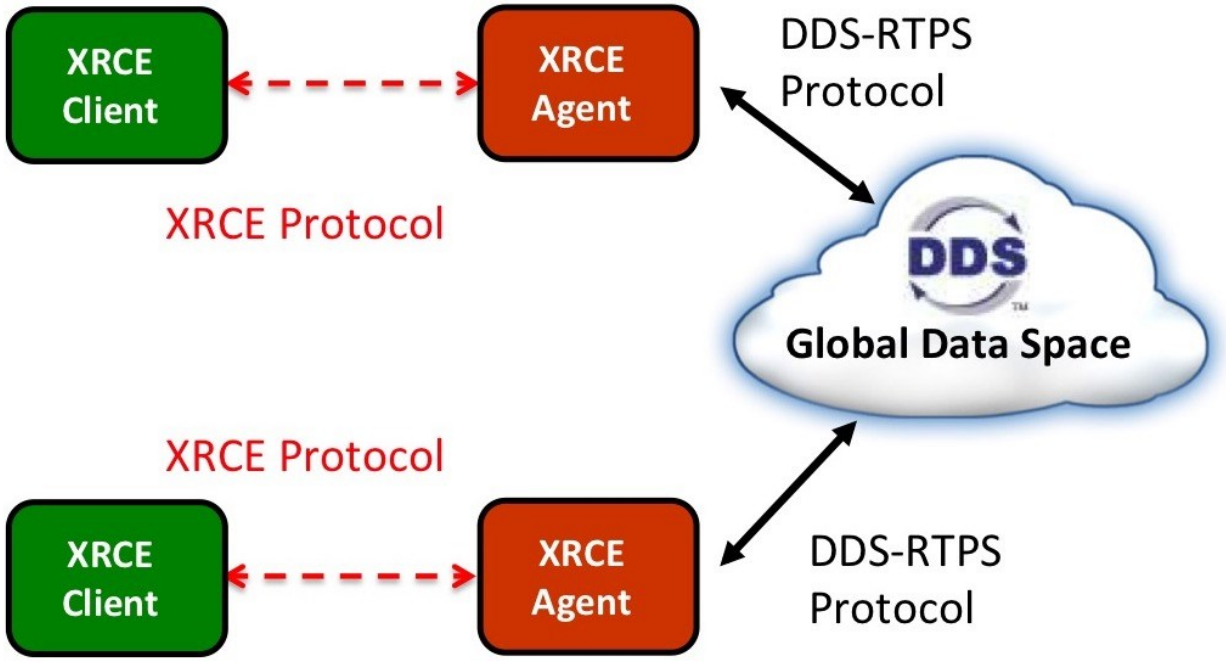


Figure 24— XRCE Agents communicating via DDS-RTPS

The XRCE Clients communicates using the XRCE Protocol with their respective XRCE Agents. Those XRCE Agents communicate with each other using DDS-RTPS, as each is a DDS DomainParticipant on the DDS Domain.

10.3 Client-to-Client communication brokered by an Agent

Multiple XRCE Client applications may be connected to the same XRCE Agent.

In it up to the implementation of the XRCE Agent whether the DDS Entities it creates are exclusive to each XRCE Client or alternatively are shared across XRCE Clients. However the behavior observable by the XRCE Client shall be as if the DDS XRCE Agent creates separate DDS Objects exclusive to each XRCE Client.

If the XRCE Agent creates separate DDS entities on behalf of each XRCE Client, then each will have its own proxy DDS DomainParticipant. These two DDS DomainParticipants will communicate with each other on the DDS Domain. In this situation the two XRCE Clients will communicate with each other "brokered" by the XRCE Agent without the need for additional configuration or logic in the XRCE Agent.

If the XRCE Agent shares DDS entities among different XRCE Clients, then the requirement to behave "as if" each had its own separate entities requires that the local DDS DataWriter entities discover and match the local DDS DataReader entities in the same DomainParticipant. This will automatically cause the XRCE Clients to communicate with each other using the Agent as a "broker" without further configuration.

An implementation of an XRCE Agent may choose to create faster communication path between the local XRCE DataWriter and DataReader objects so that data from an XRCE DataWriter can go directly to the matched XRCE DataReader without having to go via the associated DDS Entities. This "shortcut" can be implemented as an optimization as it does not impact any of the protocols nor it impacts interoperability with other XRCE Clients, Agents, or DDS DomainParticipants.

This type of scenario is shown in illustrated in Figure 25 below.

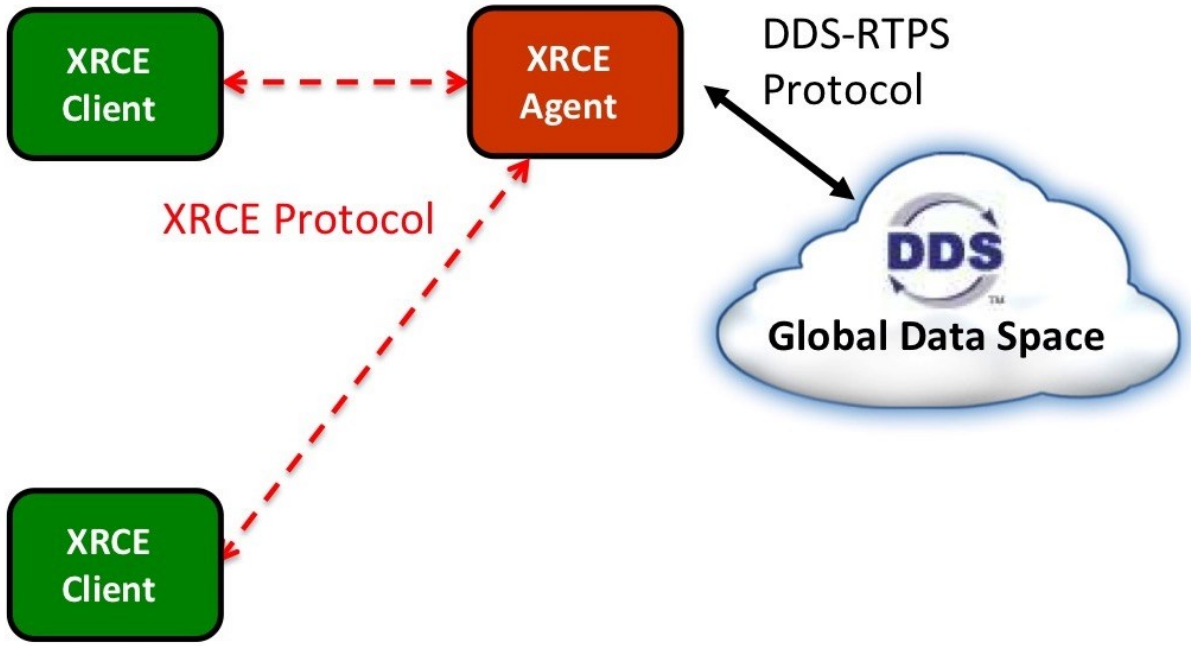


Figure 25— XRCE Clients communicating using the XRCE Agent as a broker

Multiople XRCE Clients may be connected to the same XRCE Agent. The XRCE Clients communicate with each other using the XRCE Agent as a "broker". This "client-to-client" communication may utilize the related DDS Objects, or may use an optimized path inside the Agent that shortcuts the use of the DDS Objects.

10.4 Federated deployment

The specification supports federated deployments where XRCE Agents appear as Clients to other XRCE Agents.

In other to support these deployments the XRCE Agent implementation must implement the client-side of the XRCE Protocol in addition to the server part.

Supporting this kind of deployment is an implementation decision, as it does not impact any of the protocols nor it impacts interoperability with other XRCE Clients, Agents, or DDS DomainParticipants.

This type of scenario is shown in illustrated in Figure 26 below.

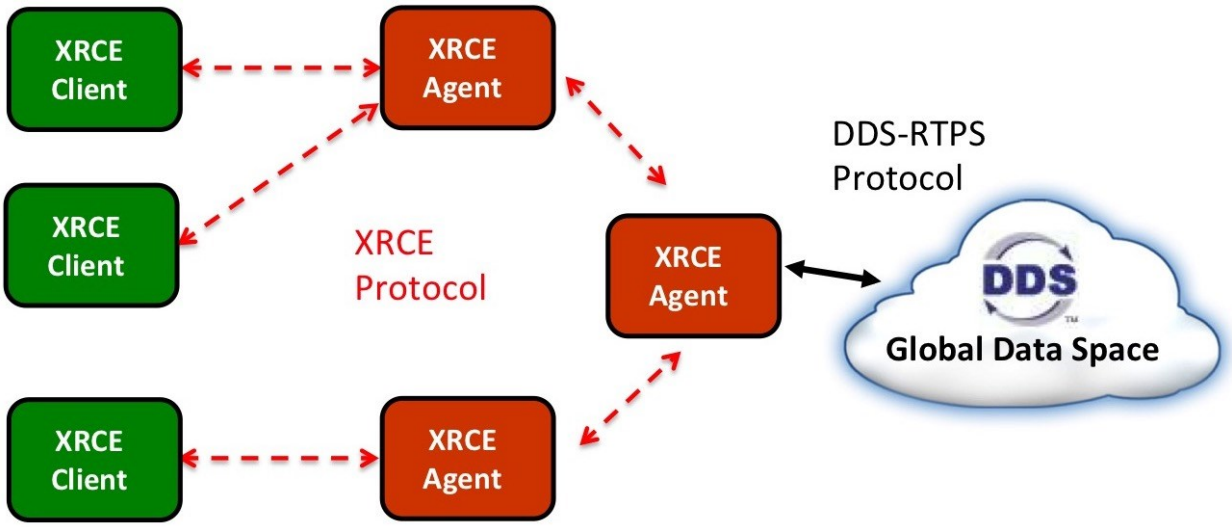


Figure 26— XRCE Agents operating as a federation

The XRCE Agents can communicate with each other using the same DDS-XRCE protocol. The Agents enable federations and store-and-forward dataflow. This type of deployment is transparent to the XRCE Client applications and the DDS applications.

10.5 Direct Peer-to-Peer communication between client Applications

The specification supports applications having direct communications using only the XRCE Protocol. In order to do this each application must implement both the XRCE Client and the XRCE Agent part of the protocol.

This deployment requires the application to create a separate XRCE Client to manage the communication with each XRCE Agent. The application would also create an XRCE Agent to manage communication with all the clients.

This deployment does not impact any of the protocols nor it impacts interoperability with other XRCE Clients, Agents, or DDS DomainParticipants.

Compared with the communication brokered by an XRCE Agent, the drawback of the direct peer-to-peer communication is that the applications need to consume more resources to instantiate the additional XRCE Clients needed to maintain the separate state with each peer XRCE Agent. Of course implementations could optimize this to nor have to create all these extra objects. However they will still need to keep separate state, especially for reliable communications.

An additional drawback of the direct peer to peer communication is that the applications cannot easily go into sleep cycles as the XRCE Agents they contain need to be active in order to process the messages from the XRCE Clients. Therefore is not suitable for many resource-constrained scenarios.

This type of scenario is shown in illustrated in Figure 27 below.

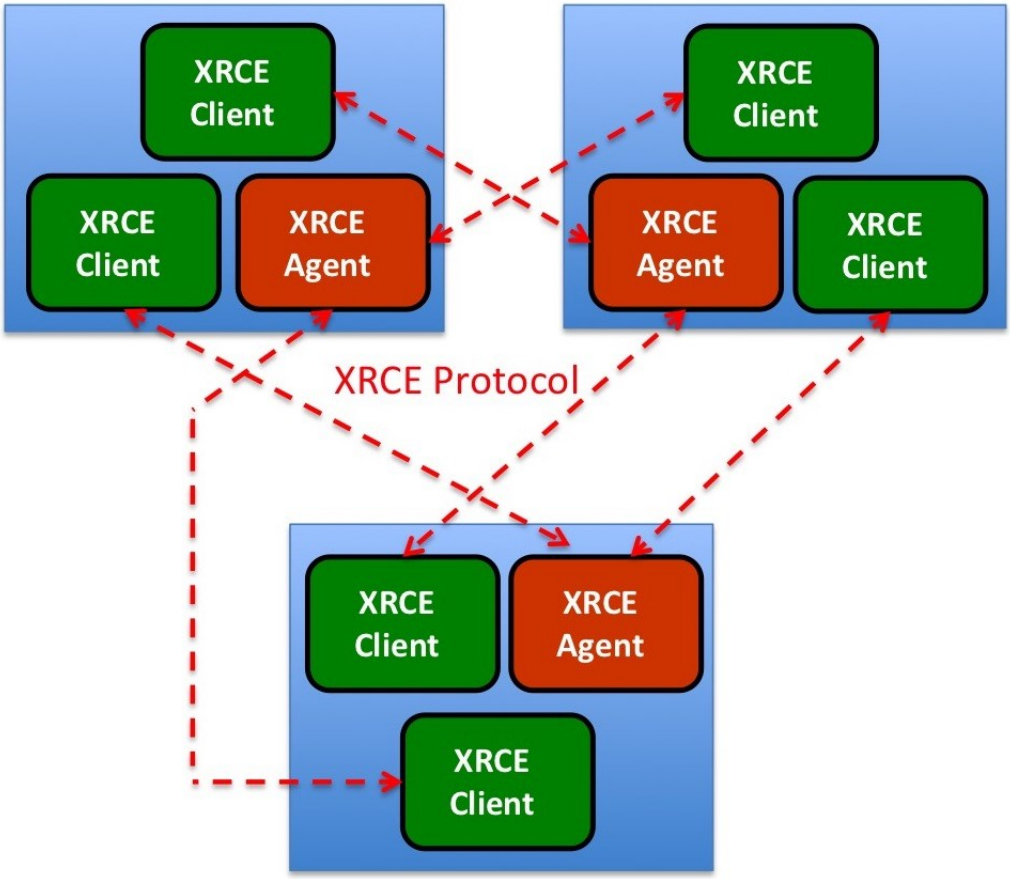


Figure 27— Direct peer-to-peer communication between XRCE Clients

Applications can communicate directly peer-to-peer without having the communication brokered by a separate XRCE Agent. To do this each Application must implement both the XRCE Client and the XRCE Agent parts of the protocol.

10.6 Combined deployment

Figure 28 below illustrates a scenario where the different deployments are combined into a single system.

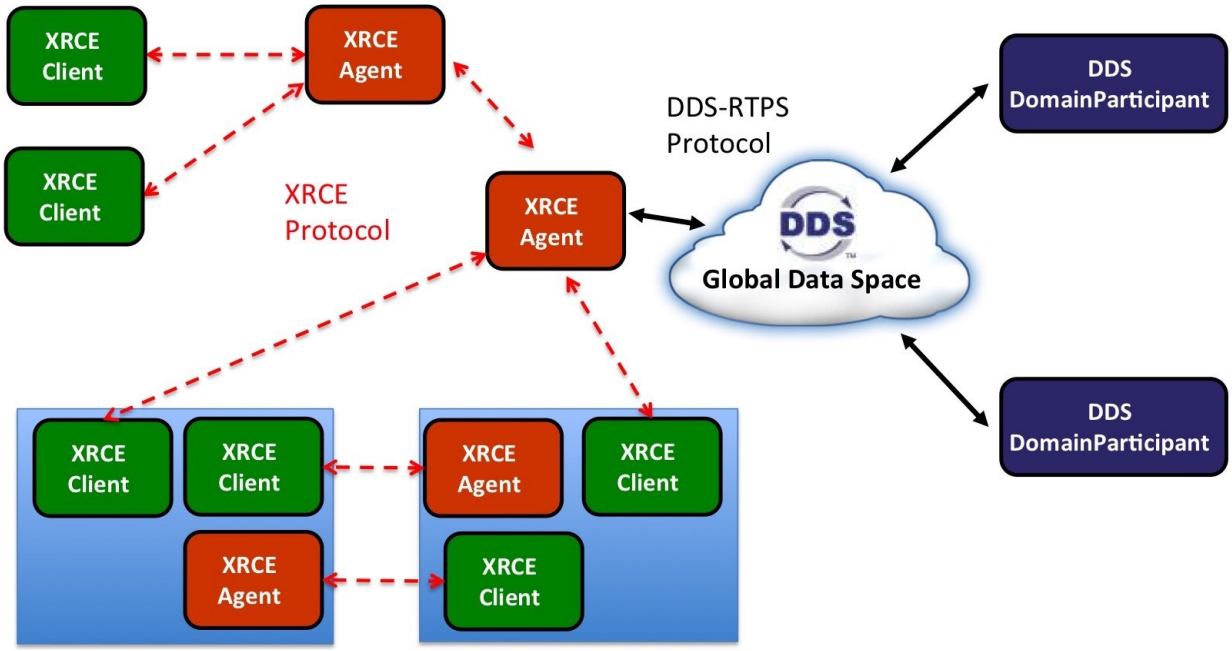


Figure 28— Combined deployment scenario

Illustrates interoperability between applications using XRCE abnd applications using DDS-RTPS. XRCE Applications may communicate via XRCE Agents acting as proxies. They can communicate peer to peer with each other using XRCE Agents as brokers or directly by implementing both the XRCE Clieant and Agent part of the protocol.

11 Transport Mappings

11.1 Transport Model

The XRCE protocol is not limited to any specific transports. It can be mapped to most existing network transports such as UDP, TCP and low bandwidth transports such as Bluetooth, ZigBee and 6LoWPAN.

To run without additional overhead it is expected that the transport supports the following functionality:

(1) Deliver messages of at least 64 bytes.
(2) Handle the integrity of messages, dropping any messages that are corrupted. This capability does not restrict the usable transports; it simply requires appending a CRC to messages from transports that do not handle integrity natively.
(3) Provide the size of the received message as well as the source address. This requirement does not restrict the usable transports; it simply requires prepending source information and size to messages from transports that do not include the information natively.
(4) Support bi-directional communication.
(5) Provide transport-level security, specifically the means for the `Client` to authenticate the `Agent` and the means for secure (encrypted and authenticated) message exchange. Alternatively the XRCE `Agent` and `Client` can be deployed on top of a secure network layer (e.g. an encrypted VPN).

The following functionality is explicitly not required from the transport:

(1) It does not need to provide reliability. Messages may be dropped.
(2) It does not need to provide ordering. Messages may arrive out of order.
(3) It does not need to provide notification of dropped messages.

Transports that do not meet some of the above pre-requisites may still be used by adding the missing information as an envelope around the XRCE message. This would be done as part of the mapping to that specific protocol.

For example is the source address or message size are missing they could be added as a prefix to the XRCE message. If the transport does not support integrity a CRC suffix could be added to the XRCE message.

11.2 UDP Transport

The UDP transport meets all the functionality listed in clause 11.1. Except that it does not provide security.

For applications requiring security there is the "Datagram Transport Layer Security" (DTLS) standard [DTLS] that provides security in top of UDP/IP. Alternatively UDP mat be deployed on a private network (VPN), which provides security at the IP layer below UDP.

Since the XRCE protocol does not require for the transport to provide reliability, ordering, or notification of failures it can be trivially mapped to "datagram" transports such as UDP/IP.

11.2.1 Transport Locators

When XRCE is mapped to the UDP v4, the `TransportLocator` union shall use the `TransportLocatorFormat` discriminator `ADDRESS_FORMAT_MEDIUM`. This selects the member ***medium_locator*** of type `TransportLocatorMedium` defined in Annex A IDL Types as:

```
struct TransportLocatorMedium {
    octet address[4];
    unsigned short port;
};
```

When XRCE is mapped to the UDP v6, the `TransportLocator` union shall use the `TransportLocatorFormat` discriminator `ADDRESS_FORMAT_LARGE`. This selects the member ***large_locator*** of type `TransportLocatorLarge` defined in Annex A IDL Types as:

```
struct TransportLocatorLarge {
    octet address[16];
    unsigned long port;
};
```

The ***address*** field shall contain the IP v6 address and the ***port*** field shall contain the UDP/IP v6 port number.

11.2.2 Connection establishment

UDP is a connectionless transport. Communication occurs between a UDP Server and a UDP Client. Each has an associated UDP/IP address and port.

- The UDP Server listens to a **server port**, which is known to the client.
- The UDP Client sends UDP datagrams to the UDP Server address and **server port**.
- The UDP Server receives the message, which includes the UDP address and port of the sending Client.
- The UDP Server sends replies back the Client using the address and port received in the message.
- The UDP Client receives replies from the server coming back to the client's address and port.

When communicating over UDP the XRCE Agent shall behave as an UDP Server and the XRCE Client as the UDP Client.

The XRCE Agent shall be pre-configured with the port number it shall listen to.

The XRCE Client shall be pre-configured with the UDP/IP address and port of the XRCE Agent.

11.2.3 Message Envelopes

The mapping of the XRCE Protocol to UDP/IP does not add any additional envelopes around the XRCE message. The UDP/IP payload shall contain exactly one XRCE message.

11.2.4 Agent Discovery

XRCE Agent discovery may be done using UDP/IP multicast. The XRCE Agents shall be pre-configured with the multicast address and port number they shall listen to. By default they shall be the address 239.255.0.2 and the port 7400.

To discover Agents via multicast the XRCE Client shall send the GET_INFO message (see 8.3.5.3) periodically to the configured multicast address and port. This message shall invoke the get_info operation (see 7.8.2.2) on the XRCE Agent, which shall respond and include its TransportLocators. The XRCE Client shall stop sending the periodic message once it receives a suitable response from an Agent.

XRCE Agent discovery may be done using UDP/IP unicast. The XRCE Clients shall be pre-configured with a list of candidate UDP addresses and ports where XRCE Agents may be located.

To discover Agents via unicast the XRCE Client shall send the GET_INFO message (see 8.3.5.3) periodically to the configured addresses and ports. This message shall invoke the get_info operation (see 7.8.2.2) on the XRCE Agent, which shall respond and include its TransportLocators. The XRCE Client shall stop sending the periodic message once it receives a suitable response from an Agent.

11.3 TCP Transport

The TCP transport meets all the functionality listed in clause 11.1. except it does not provide security nor information on the message size.

For applications requiring security there is the “Transport Layer Security (TLS)” standard [TLS] that provides security in top of TCP/IP. Alternatively TCP/IP mat be deployed on a private network (VPN), which provides security at the IP layer below TCP.

The message size shall be added as a prefix ahead of the XRCE message as defined in 11.3.3.

11.3.1 Transport Locators

When XRCE is mapped to the UDP/IP version 4 , the `TransportLocator` union shall use the `TransportLocatorFormat` discriminator `ADDRESS_FORMAT_MEDIUM`. This selects the member ***medium_locator*** of type `TransportLocatorMedium` defined in Annex A IDL Types as:

```
struct TransportLocatorMedium {
    octet address[4];
    unsigned short port;
};
```

When XRCE is mapped to the TCP/IP version 6, the `TransportLocator` union shall use the `TransportLocatorFormat` discriminator `ADDRESS_FORMAT_LARGE`. This selects the member ***large_locator*** of type `TransportLocatorLarge` defined in Annex A IDL Types as:

```
struct TransportLocatorLarge {
    octet address[16];
    unsigned long port;
};
```

The ***address*** field shall contain the IP v6 address and the ***port*** field shall contain the TCP/IP v6 port number.

11.3.2 Connection establishment

TCP is a connection-oriented transport. Communication occurs between a TCP Client and a TCP Server. Each has an associated TCP/IP address and port.

- The TCP Server listens to a **server port**, which is known to the client.
- The TCP Client connects to the Server.
- The TCP Server accepts the connection from the Client. This establishes a bi-directional communication channel. Both ends can send and receive o that channel.
- The TCP Client can send and receive messages to and from the Server.
- The TCP Server can send and receive messages to and from the Client.

When communicating over TCP the XRCE Agent shall behave as a TCP Server and the XRCE Client as the TCP Client.

The XRCE Agent shall be pre-configured with the port number it shall listen to.

The XRCE Client shall be pre-configured with the TCP/IP address and port of the XRCE Agent.

11.3.3 Message Envelopes

The mapping of the XRCE Protocol to TCP/IP adds a 2-byte prefix as an envelope the XRCE message. The 2-byte prefix shall contain the length of the XRCE message that follows encoded as little endian.

After the 2-byte envelope the TCP/IP payload shall contain exactly one XRCE message. The alignment of the XRCE message shall not be changed by the added 2-byte prefix. Stated differently the XRCE message shall consider its first byte to be aligned to an 8-byte (XCDR maximum alignment) boundary.

11.3.4 Agent Discovery

XRCE Agent discovery may be done using UDP/IP multicast even if the communication will be over TCP.

The XRCE Agents may be pre-configured with the multicast address and port number they shall listen to. By default they shall be the address 239.255.0.2 and the port 7400.

To discover Agents via multicast the XRCE Client shall send the GET_INFO message (see 8.3.5.3) periodically to the configured multicast address and port. This message shall invoke the get_info operation (see 7.8.2.2) on the XRCE Agent, which shall respond and include its TransportLocators. The XRCE Client shall stop sending the periodic message once it receives a suitable response from an Agent.

XRCE Agent discovery may be done using TCP/IP. The XRCE Clients shall be pre-configured with a list of candidate TCP addresses and ports where XRCE Agents may be located.

To discover Agents via unicast the XRCE Client shall periodically attempt to establish TCP connections to the configured addresses and ports. One a connection is established it shall send the CREATE_CLIENT message (see 8.3.5.1). This message shall invoke the create_client operation (see 7.8.2.1) on the XRCE Agent, which shall either accept or produce an error. The XRCE Client shall stop making periodic connection attempts once it receives a suitable response from an Agent.

11.4 Other Transports

The XRCE Protocol is well suited to be mapped to other transports, even transport with small bandwidth and MTUs such as IEEE 802.15.4, Zigbee, Bluetooth, and 6LoWPAN.

The fact that the XRCE Protocol has minimal requirements on the transport (i.e. does not require ordering or reliable delivery), provides the means for authentication, and can do its own data fragmentation and re-assembly means that most transports mappings can simply include the XRCE message as a payload without additional envelopes.

However, in order to get transparent interoperability between vendors it is required to define the precise encoding of the transport locators as well as the means to discover agents and establish initial communicators. Therefore it is expected that future revisions of this specification will provide additional transport mappings.

A IDL Types

```
module dds { module xrce {

    typedef octet ClientKey[4];

// IDL does not have a syntax to express array constants so we
// use #define with is legal in IDL
#define  CLIENTKEY_INVALID {0x00, 0x00, 0x00, 0x00}

    typedef octet ObjectKind;

    const ObjectKind OBJK_INVALID     = 0x00;
    const ObjectKind OBJK_PARTICIPANT = 0x01;
    const ObjectKind OBJK_TOPIC       = 0x02;
    const ObjectKind OBJK_PUBLISHER   = 0x03;
    const ObjectKind OBJK_SUBSCRIBER  = 0x04;
    const ObjectKind OBJK_DATAWRITER  = 0x05;
    const ObjectKind OBJK_DATAREADER  = 0x06;
    const ObjectKind OBJK_TYPE        = 0x0A;
    const ObjectKind OBJK_QOSPROFILE  = 0x0B;
    const ObjectKind OBJK_APPLICATION = 0x0C;
    const ObjectKind OBJK_AGENT       = 0x0D;
    const ObjectKind OBJK_CLIENT      = 0x0E;
    const ObjectKind OBJK_OTHER       = 0x0F;

    typedef octet ObjectId     [2];
    typedef octet ObjectPrefix [2];

    // There are three predefined values ObjectId
    // IDL does not have a syntax to express array constants so we
    // use #define with is legal in IDL
#define  OBJECTID_INVALID {0x00,0x00}
#define  OBJECTID_AGENT   {0xFF,0xFD}
#define  OBJECTID_CLIENT  {0xFF,0xFE}
#define  OBJECTID_SESSION {0xFF,0xFF}
```

```
    typedef octet XrceCookie[4];
    // Spells 'X' 'R' 'C' 'E'
#define XRCE_COOKIE { 0x58, 0x52, 0x43, 0x45 }

    typedef octet XrceVersion[2];
#define XRCE_VERSION_MAJOR     0x01
#define XRCE_VERSION_MINOR     0x00
#define XRCE_VERSION  { XRCE_VERSION_MAJOR, XRCE_VERSION_MINOR }

    typedef octet XrceVendorId[2];
#define XRCE_VENDOR_INVALID1  0x00
#define XRCE_VENDOR_INVALID1  0x00


    struct Time_t {
        long           seconds;
        unsigned long  nanoseconds;
    };

    typedef octet SessionId;
    const SessionId SESSIONID_NONE_WITH_CLIENT_KEY     = 0x00;
    const SessionId SESSIONID_NONE_WITHOUT_CLIENT_KEY  = 0x80;

    typedef octet StreamId;
    const SessionId STREAMID_NONE                        = 0x00;
    const SessionId STREAMID_BUILTIN_BEST_EFFORTS        = 0x01;
    const SessionId STREAMID_BUILTIN_RELIABLE            = 0x80;

    @bit_bound(8)
    enum TransportLocatorFormat {
        ADDRESS_FORMAT_SMALL,
        ADDRESS_FORMAT_MEDIUM,
        ADDRESS_FORMAT_LARGE,
        ADDRESS_FORMAT_STRING
    };

    struct TransportLocatorSmall {
```

```
    octet address[2];
    octet locator_port;
};
struct TransportLocatorMedium {
    octet address[4];
    unsigned short locator_port;
};
struct TransportLocatorLarge {
    octet address[16];
    unsigned long locator_port;
};
struct TransportLocatorString {
    string value;
};

union TransportLocator switch (TransportLocatorFormat) {
    case ADDRESS_FORMAT_SMALL:
    TransportLocatorSmall small_locator;
    case ADDRESS_FORMAT_MEDIUM:
    TransportLocatorMedium medium_locator;
    case ADDRESS_FORMAT_LARGE:
    TransportLocatorLarge medium_locator;
    case ADDRESS_FORMAT_STRING:
    TransportLocatorString string_locator;
};
typedef sequence<TransportLocator> TransportLocatorSeq;

struct Property {
    string name;
    string value;
};
typedef sequence<Property> PropertySeq;

@extensibility(FINAL)
struct CLIENT_Representation {
    XrceCookie   xrce_cookie;  // XRCE_COOKIE
    XrceVersion  xrce_version;
    XrceVendorId xrce_vendor_id;
```

```
    Time_t       client_timestamp;
    ClientKey    client_key;
    SessionId    session_id;
    @optional  PropertySeq properties;
};

@extensibility(FINAL)
struct AGENT_Representation {
    XrceCookie   xrce_cookie;  // XRCE_COOKIE
    XrceVersion  xrce_version;
    XrceVendorId xrce_vendor_id;
    Time_t       agent_timestamp;
    @optional  PropertySeq  properties;
};

typedef octet RepresentationFormat;
const RepresentationFormat REPRESENTATION_BY_REFERENCE  = 0x01;
const RepresentationFormat REPRESENTATION_AS_XML_STRING = 0x02;
const RepresentationFormat REPRESENTATION_IN_BINARY     = 0x03;

const long REFERENCE_MAX_LEN  = 128;

@extensibility(FINAL)
union OBJK_Representation3Formats switch(RepresentationFormat) {
  case REPRESENTATION_BY_REFERENCE :
    string<REFERENCE_MAX_LEN>  object_reference;
  case REPRESENTATION_AS_XML_STRING :
    string           xml_string_representation;
  case REPRESENTATION_IN_BINARY :
    sequence<octet>  binary_representation;
};

@extensibility(FINAL)
union OBJK_RepresentationRefAndXMLFormats switch(RepresentationFormat) {
  case REPRESENTATION_BY_REFERENCE :
    string<REFERENCE_MAX_LEN>  object_reference;
  case REPRESENTATION_AS_XML_STRING :
```

```
    string           string_representation;
};

@extensibility(FINAL)
union OBJK_RepresentationBinAndXMLFormats switch(RepresentationFormat) {
  case REPRESENTATION_IN_BINARY :
    sequence<octet>  binary_representation;
  case REPRESENTATION_AS_XML_STRING :
    string           string_representation;
};

@extensibility(FINAL)
struct OBJK_RepresentationRefAndXML_Base {
  OBJK_RepresentationRefAndXMLFormats representation;
};

@extensibility(FINAL)
struct OBJK_RepresentationBinAndXML_Base {
  OBJK_RepresentationBinAndXMLFormats representation;
};

@extensibility(FINAL)
struct OBJK_Representation3_Base {
  OBJK_Representation3Formats representation;
};

/* Objects supporting by Reference and XML formats */

@extensibility(FINAL)
struct OBJK_QOSPROFILE_Representation : OBJK_RepresentationRefAndXML_Base {
};

@extensibility(FINAL)
struct OBJK_TYPE_Representation       : OBJK_RepresentationRefAndXML_Base {
};

@extensibility(FINAL)
struct OBJK_DOMAIN_Representation : OBJK_RepresentationRefAndXML_Base {
```

```
};

@extensibility(FINAL)
struct OBJK_APPLICATION_Representation : OBJK_RepresentationRefAndXML_Base {
};


/* Objects supporting Binary and XML formats */
@extensibility(FINAL)
struct OBJK_PUBLISHER_Representation   : OBJK_RepresentationBinAndXML_Base {
    ObjectId participant_id;
};

@extensibility(FINAL)
struct OBJK_SUBSCRIBER_Representation : OBJK_RepresentationBinAndXML_Base {
    ObjectId participant_id;
};

@extensibility(FINAL)
struct DATAWRITER_Representation    : OBJK_RepresentationBinAndXML_Base {
    ObjectId publisher_id;
};

@extensibility(FINAL)
struct DATAREADER_Representation    : OBJK_RepresentationBinAndXML_Base {
    ObjectId subscriber_id;
};

/* Objects supporting all 3 representation formats */
@extensibility(FINAL)
struct OBJK_PARTICIPANT_Representation : OBJK_Representation3_Base {
    short     domain_id;
};

@extensibility(FINAL)
struct OBJK_TOPIC_Representation : OBJK_Representation3_Base {
    ObjectId participant_id;
};
```

```
/* Binary representations */
@extensibility(APPENDABLE)
struct  OBJK_DomainParticipant_Binary {
    @optional string<128> domain_reference;
    @optional string<128> qos_profile_reference;
};

@extensibility(APPENDABLE)
struct  OBJK_Topic_Binary {
    string<256> topic_name;
    @optional string<256> type_reference;
    @optional DDS:XTypes::TypeIdentifier type_identifier;
};

@extensibility(FINAL)
struct  OBJK_Publisher_Binary_Qos {
    @optional sequence<string>  partitions;
    @optional sequence<octet>   group_data;
};

@extensibility(APPENDABLE)
struct  OBJK_Publisher_Binary {
    @optional string publisher_name;
    @optional OBJK_Publisher_Binary_Qos  qos;
};

@extensibility(FINAL)
struct  OBJK_Subscriber_Binary_Qos {
    @optional sequence<string>  partitions;
    @optional sequence<octet>   group_data;
};

@extensibility(APPENDABLE)
struct  OBJK_Subscriber_Binary {
    @optional string subscriber_name;
    @optional OBJK_Subscriber_Binary_Qos  qos;
```

```
};

@bit_bound(16)
bitmask EndpointQosFlags {
    @position(0) is_reliable,
    @position(1) is_history_keep_all,
    @position(2) is_ownership_exclusive,
    @position(3) is_durability_transient_local,
    @position(4) is_durability_transient,
    @position(5) is_durability_persistent,
};

@extensibility(FINAL)
struct  OBJK_Endpoint_Binary_Qos {
    EndpointQosFlags            qos_flags;
    @optional unsigned short    history_depth;
    @optional unsigned long     deadline_msec;
    @optional unsigned long     lifespan_msec;
    @optional sequence<octet>   user_data;
};

@extensibility(FINAL)
struct  OBJK_DataWriter_Binary_Qos :  OBJK_Endpoint_Binary_Qos {
    @optional unsigned long     ownership_strength;
};

@extensibility(FINAL)
struct  OBJK_DataReader_Binary_Qos :  OBJK_Endpoint_Binary_Qos {
    @optional unsigned long     timebasedfilter_msec;
    @optional string            contentbased_filter;
};

@extensibility(APPENDABLE)
struct  OBJK_DataReader_Binary {
    string                                  topic_name;
    @optional OBJK_DataReader_Binary_Qos    qos;
};
```

```
@extensibility(APPENDABLE)
struct  OBJK_DataWriter_Binary {
    string                                topic_name;
    @optonal OBJK_DataWriter_Binary_Qos  qos;
};

@extensibility(FINAL)
union ObjectVariant switch(ObjectKind) {
  // case OBJK_INVALID : indicates default or selected by Agent. No data.
case OBJK_AGENT :
  AGENT_Representation client;
case OBJK_CLIENT :
  CLIENT_Representation client;
case OBJK_APPLICATION :
  OBJK_APPLICATION_Representation application;
case OBJK_PARTICIPANT :
  OBJK_PARTICIPANT_Representation participant;
case OBJK_QOSPROFILE :
  OBJK_QOSPROFILE_Representation qos_profile;
case OBJK_TYPE :
  OBJK_TYPE_Representation type;
case OBJK_TOPIC :
  OBJK_TOPIC_Representation topic;
case OBJK_PUBLISHER :
  OBJK_PUBLISHER_Representation publisher;
case OBJK_SUBSCRIBER :
  OBJK_SUBSCRIBER_Representation subscriber;
case OBJK_DATAWRITER :
  DATAWRITER_Representation data_writer;
case OBJK_DATAREADER :
  DATAREADER_Representation data_reader;
};

struct CreationMode {
  boolean reuse;
  boolean replace;
};
```

```
typedef octet RequestId[2];

@bit_bound(8)
enum StatusValue {
    @value(0x00) STATUS_OK,
    @value(0x01) STATUS_OK_MATCHED,
    @value(0x80) STATUS_ERR_DDS_ERROR,
    @value(0x81) STATUS_ERR_MISMATCH,
    @value(0x82) STATUS_ERR_ALREADY_EXISTS,
    @value(0x83) STATUS_ERR_DENIED,
    @value(0x84) STATUS_ERR_UNKNOWN_REFERENCE,
    @value(0x85) STATUS_ERR_INVALID_DATA,
    @value(0x86) STATUS_ERR_INCOMPATIBLE,
    @value(0x87) STATUS_ERR_RESOURCES
};

struct ResultStatus {
  StatusValue  status;
  octet        implementation_status;
};

const   octet STATUS_LAST_OP_NONE      = 0;
const   octet STATUS_LAST_OP_CREATE    = 1;
const   octet STATUS_LAST_OP_UPDATE    = 2;
const   octet STATUS_LAST_OP_DELETE    = 3;
const   octet STATUS_LAST_OP_LOOKUP    = 4;
const   octet STATUS_LAST_OP_READ      = 5;
const   octet STATUS_LAST_OP_WRITE     = 6;

bitmask InfoMask {
  @position(0) INFO_CONFIGURATION,
  @position(1) INFO_ACTIVITY
};

@extensibility(APPENDABLE)
struct AGENT_ActivityInfo {
    short availability;
```

```
    TransportLocatorSeq address_seq;
};

@extensibility(APPENDABLE)
struct DATAREADER_ActivityInfo {
    short highest_acked_num;
};

@extensibility(APPENDABLE)
struct DATAWRITER_ActivityInfo {
    unsigned long long sample_seq_num;
    short stream_seq_num;

};

@extensibility(FINAL)
union ActivityInfoVariant switch (ObjectKind) {
  case OBJK_DATAWRITER :
    DATAWRITER_ActivityInfo data_writer;
  case OBJK_DATAREADER :
    DATAREADER_ActivityInfo data_reader;
};

@extensibility(FINAL)
struct ObjectInfo {
    @optional  ActivityInfoVariant  activity;
    @optional  ObjectVariant        config;

};

@extensibility(FINAL)
struct BaseObjectRequest {
  RequestId     request_id;
  ObjectId      object_id;
};

typedef RelatedObjectRequest BaseObjectRequest;
```

```
@extensibility(FINAL)
struct BaseObjectReply {
  BaseObjectRequest  related_request;
  ResultStatus       result;
};

typedef octet DataFormat;
const DataFormat FORMAT_DATA           = 0x00; // 0b0000 0000
const DataFormat FORMAT_SAMPLE         = 0x02; // 0b0000 0010
const DataFormat FORMAT_DATA_SEQ       = 0x08; // 0b0000 1000
const DataFormat FORMAT_SAMPLE_SEQ     = 0x0A; // 0b0000 1010
const DataFormat FORMAT_PACKED_SAMPLES = 0x0E; // 0b0000 1110
const DataFormat FORMAT_MASK           = 0x0E; // 0b0000 1110

@extensibility(APPENDABLE)
struct DataDeliveryControl {
    unsigned short max_samples;
    unsigned short max_elapsed_time;
    unsigned short max_bytes_per_second;
    unsigned short min_pace_period;  // milliseconds
};

@extensibility(FINAL)
struct ReadSpecification  {
    DataFormat data_format;
    @optional string              content_filter_expression;
    @optional DataDeliveryControl delivery_control;
};

@bit_bound(8)
bitmask SampleInfoFlags {
    @position(0) INSTANCE_STATE_UNREGISTERED,
    @position(1) INSTANCE_STATE DISPOSED,
    @position(2) VIEW_STATE_NEW,
    @position(3) SAMPLE_STATE_READ,
};

typedef octet SampleInfoFormat;
```

```
const SampleInfoFormat FORMAT_EMPTY      = 0x00; // 0b0000 0000
const SampleInfoFormat FORMAT_SEQNUM     = 0x01; // 0b0000 0001
const SampleInfoFormat FORMAT_TIMESTAMP  = 0x02; // 0b0000 0010
const SampleInfoFormat FORMAT_SEQN_TIMS  = 0x03; // 0b0000 0011

@extensibility(FINAL)
struct SeqNumberAndTimestamp {
    unsigned long   sequence_number;
    unsigned long   session_time_offset; // milliseconds up to 53 days
};

@extensibility(FINAL)
union SampleInfoDetail switch(SampleInfoFormat) {
  case FORMAT_EMPTY:
  case FORMAT_SEQNUM:
    unsigned long   sequence_number;
  case FORMAT_TIMESTAMP:
    unsigned long   session_time_offset; // milliseconds up to 53 days
  case FORMAT_TIMESTAMP:
    SeqNumberAndTimestamp seqnum_n_timestamp;
};

@extensibility(FINAL)
struct SampleInfo {
    SampleInfoFlags  state; //Combines SampleState, InstanceState, ViewState
    SampleInfoDetail detail;
};

typedef unsigned short  DeciSecond; // 10e-1 seconds
@extensibility(FINAL)
struct SampleInfoDelta {
  SampleInfoFlags state;  // Combines SampleState, InstanceState, ViewState
  octet           seq_number_delta;
  DeciSecond      timestamp_delta; // In 1/10 of seconds
};

@extensibility(FINAL)
struct SampleData {
```

```
    XCDRSerializedBuffer serialized_data;
};
typedef sequence<SampleData> SampleDataSeq;

@extensibility(FINAL)
struct Sample {
  SampleInfo   info;
  SampleData   data;
};
typedef sequence<Sample> SampleSeq;

@extensibility(FINAL)
struct SampleDelta {
  SampleInfoDelta   info_delta;
  SampleData        data;
};

@extensibility(FINAL)
struct PackedSamples {
  SampleInfo             info_base;
  sequence<SampleDelta>  sample_delta_seq;
};
typedef sequence<SamplePacked> SamplePackedSeq;

@extensibility(FINAL)
union DataRepresentation switch(DataFormat) {
  case FORMAT_DATA:
    SampleData data;
  case FORMAT_SAMPLE:
    Sample sample;
  case FORMAT_DATA_SEQ:
    SampleDataSeq data_seq;
  case FORMAT_SAMPLE_SEQ:
    SampleSeq sample_seq;
  case FORMAT_PACKED_SAMPLES:
    PackedSamples packed_samples;
};
```

```
// Message Payloads
@extensibility(FINAL)
struct CREATE_CLIENT_Payload : BaseObjectRequest {
    CLIENT_Representation client_representation;
};

@extensibility(FINAL)
struct CREATE_Payload : BaseObjectRequest {
    ObjectVariant  object_representation;
};

@extensibility(FINAL)
struct GET_INFO_Payload : BaseObjectRequest {
    InfoMask  info_mask;
};

@extensibility(FINAL)
struct DELETE_Payload : BaseObjectRequest {
};

@extensibility(FINAL)
struct STATUS_AGENT_Payload : BaseObjectReply {
    AGENT_Representation agent_info;
};

@extensibility(FINAL)
struct STATUS_Payload : BaseObjectReply {
};

@extensibility(FINAL)
struct INFO_Payload : BaseObjectReply {
    ObjectInfo     object_info;
};

@extensibility(FINAL)
struct READ_DATA_Payload : BaseObjectRequest {
    ReadSpecification               read_specification;
};
```

```
// There are 5 types of DATA and WRITE_DATA payloads
@extensibility(FINAL)
struct WRITE_DATA_Payload_Data : BaseObjectRequest {
    SampleData              data;
};

@extensibility(FINAL)
struct WRITE_DATA_Payload_Sample : BaseObjectRequest {
    Sample                sample;
};

@extensibility(FINAL)
struct WRITE_DATA_Payload_DataSeq : BaseObjectRequest {
    sequence<SampleData>    data_seq;
};

@extensibility(FINAL)
struct WRITE_DATA_Payload_SampleSeq : BaseObjectRequest {
    sequence<Sample>        sample_seq;
};

@extensibility(FINAL)
struct WRITE_DATA_Payload_PackedSamples : BaseObjectRequest {
    PackedSamples           packed_samples;
};

@extensibility(FINAL)
struct DATA_Payload_Data : RelatedObjectRequest {
    SampleData              data;
};

@extensibility(FINAL)
struct DATA_Payload_Sample : RelatedObjectRequest {
      Sample                sample;
};

@extensibility(FINAL)
```

```
struct DATA_Payload_DataSeq : RelatedObjectRequest {
    sequence<SampleData>    data_seq;
};

@extensibility(FINAL)
struct DATA_Payload_SampleSeq : RelatedObjectRequest {
    sequence<Sample>        sample_seq;
};

@extensibility(FINAL)
struct DATA_Payload_PackedSamples : RelatedObjectRequest {
    PackedSamples           packed_samples;
};

struct ACKNACK_Payload {
    short  first_unacked_seq_num;
    octet  nack_bitmap[2];
};

@extensibility(FINAL)
struct HEARTBEAT_Payload {
    short  first_unacked_seq_nr;
    short  last_unacked_seq_nr;
};

@bit_bound(8)
enum SubmessageId {
    @value(0)  CREATE_CLIENT,
    @value(1)  CREATE,
    @value(2)  GET_INFO,
    @value(3)  DELETE,
    @value(4)  STATUS,
    @value(5)  INFO,
    @value(6)  WRITE_DATA,
    @value(7)  READ_DATA,
    @value(8)  DATA,
    @value(9)  ACKNACK,
    @value(10) HEARTBEAT,
```

```
        @value(12) FRAGMENT,
        @value(13) FRAGMENT_END
    };


}; };
```

B Example Messages (Non-Normative)

B.1. CREATE_CLIENT message example

The following message could be used by a XRCE `Client` request a XRCE `ProxyClient` to be created.

The `Client` is from ***vendor_id*** {0x0F, 0x0F} and is using ***xrce_version*** {0x01, 0x00}.

The ***request_id*** is {0xAA, 0x00}, the ***client_timestamp*** is {1518905996 , 500000000} (in hexadecimal {0x5A88AA8C, 0x1DCD6500}), the ***client_key*** is {0x22, 0x33, 0x44, 0x55} and the requested ***session_id*** is 0xDD.

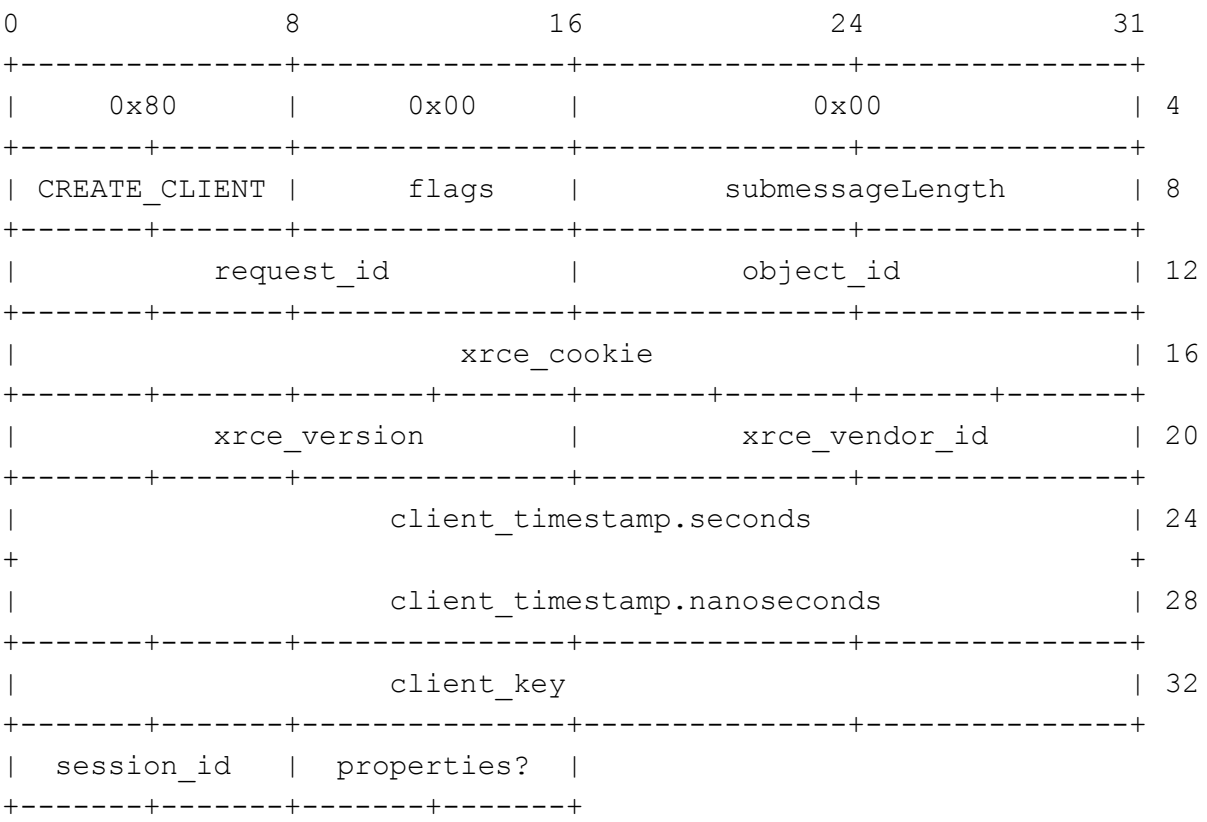


Table 19 describes each of the bytes in the message.

Table 19 Description of the CREATE_CLIENT example bytes

Bytes		Description
0-3	Message Header	
	Byte 0	sessionId = 0x80 = SESSION_ID_NONE_WITHOUT_CLIENT_KEY Indicates that there is no session and that the ***client_key*** does not follow the Message `Header`, see 8.3.2.1.
	Byte 1	streamId = 0x00 = STREAMID_NONE

		Indicates there is no stream see 8.3.2.2
	Bytes 2-3	sequenceNr = 0
4-7	Submessage Header	
	Byte 4	submessageId = CREATE_CLIENT = 0x00 See 8.3.5
	Byte 5	flags = 0x07 (reuse, replace, little endian)
	Bytes 6-7	submessageLength = 26 = 0x001B Represented in little endian as {0x1B, 0x00}
8-32	CREATE_CLIENT_Payload	
	Bytes 8-11 used for BaseObjectRequest	
	Bytes 8-9	request_id = {0xAA , 0x00}
	Bytes 10-11	object_id = {0xFF, 0xFE} Set to OBJECTID_CLIENT, see 7.6.
	Bytes 12-32 used for the CLIENT_Representation	
	Bytes 12-15	xrce_cookie = { ‘X’, ‘R’, ‘C’, ‘E’ }
	Bytes 16-17	xrce_version = {0x01, 0x00}
	Bytes 18-19	xrce_vendor_id = {0x0F, 0x0F]
	Bytes 20-27	client_timestamp = {0x5A88AA8C, 0x1DCD6500} Since flags has the Endianness bit set to 1 the timestamp is represented as little endian as { {0x8C, 0xAA, 0x99, 0x5A}, {0x00, 0x65, 0xCD, 0x1D} }
	Bytes 28-31	client_key = {0x22, 0x33, 0x44, 0x55}
	Byte 32	The requested session_id = 0xDD
	Byte 33	properties? = FALSE Indicates that the optional field ***properties*** is not present.

B.2. CREATE message examples

B.2.1. Create a DomainParticipant using REPRESENTATION_BY_REFERENCE

The following message would be used by a XRCE Client request a XRCE ProxyClient to create an XRCE DomainParticipant with ObjectID {0xDD, 0xD1} with preconfigured entities and Qos.

The `DomainParticipant` is represented by a reference to a pre-configured definition known to the XRCE Agent. Therefore the RepresentationFormat is set to REPRESENTATION_BY_REFERENCE.

The representation by reference uses a string containing the fully qualified name of DomainParticipant. See 7.7.3.6.1. In this example the reference is “MyLibrary::MyParticipant”:

The corresponding message is:

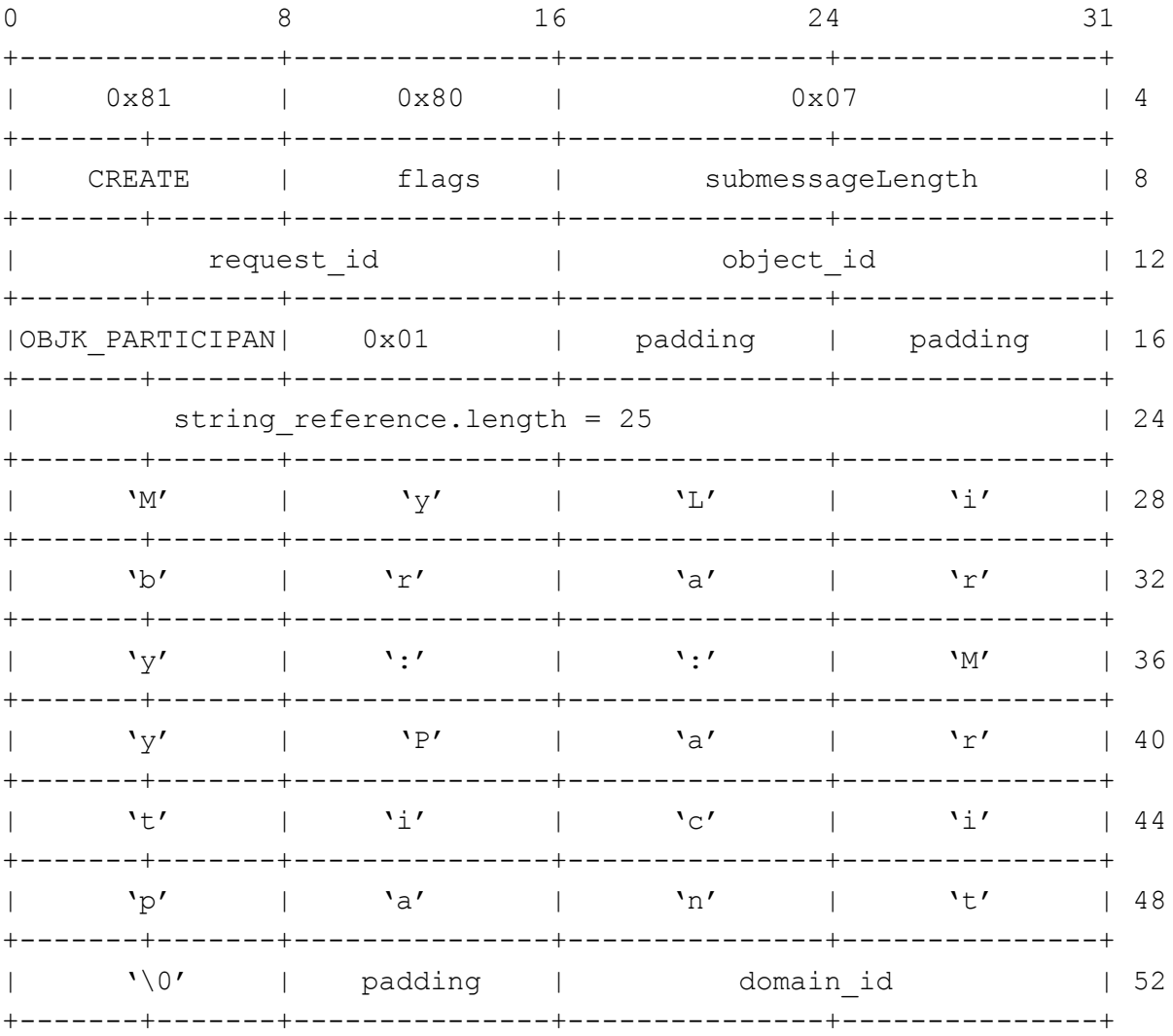


Table 22 describes the bytes in the ***CREATE*** message.

Table 20 Description of the CREATE message for the DomainParticipant using a string representation

Bytes		Description
0-3	Message Header	
	Byte 0	sessionId = 0x81 Indicates session 1 with no client key included in the message.
	Byte 1	streamId=0x80 Selects the builtin reliable stream, see 8.3.2.2
	Bytes 2-3	sequenceNr = 0x07
4-7	Submessage Header	
	Byte 4	submessageId = CREATE = 0x01 See 8.3.5.2
	Byte 5	flags = 0x07 (reuse, replace, little endian)
	Bytes 6-7	submessageLength = 26 Represented in little endian as {0x1A, 0x00}
8-51	CREATE_Payload	
	Bytes 8-11 used for BaseObjectRequest (base class of CREATE_Payload)	
	Bytes 8-9	BaseObjectRequest request_id = {0xAA , 0x01
	Bytes 10-11	BaseObjectRequest object_id = {0xDD, 0xD1} For a description of the ObjectID see 7.6.
	Bytes 12-32 used for the ObjectVariant	
	Byte 12	ObjectVariant discriminator = 0x01 Set to OBJK_PARTICIPANT
	Bytes 13-32 are OBJK_Representation3_Base (base class of PARTICIPANT_Representation)	
	Byte 13	OBJK_Representation3_Base discriminator = 0x01 RepresentationFormat set to REPRESENTATION_BY_REFERENCE
	Bytes 14-15	padding

	Bytes 16-19	string_representation.length = 25 = 0x19 Encodes length of the string represented in little endian as {0x19, 0x00, 0x00, 0x00}
	Bytes 24-48	Characters of the string_repreentation, including the terminating NUL. Total of 25 characters
	Byte 49	padding
	Bytes 50-51 used for the PARTICIPANT_Representation beyond its base class	
	Bytes 50-51	domain_id = {0x00, 0x00} Little endian representation of domain_id 0.

B.2.2. Create a DomainParticipant using REPRESENTATION_IN_BINARY

The following message would be used by a XRCE Client request a XRCE ProxyClient to create an XRCE DomainParticipant with ObjectID {0xDD, 0xD1} using default Qos.

The `DomainParticipant` is represented in binary. Therefore the RepresentationFormat is set to REPRESENTATION_IN_BINARY. In this example it will use little endian encoding.

The binary representation of a `DomainParticipant` uses the XCDR serialized representation of the type `OBJK_DomainParticipant_Binary` defined in Annex A IDL Types as:

```
@extensibility(APPENDABLE)
struct  OBJK_DomainParticipant_Binary {
  @optional string<128> domain_reference;
  @optional string<128> qos_profile_reference;
};
```

The corresponding message is:

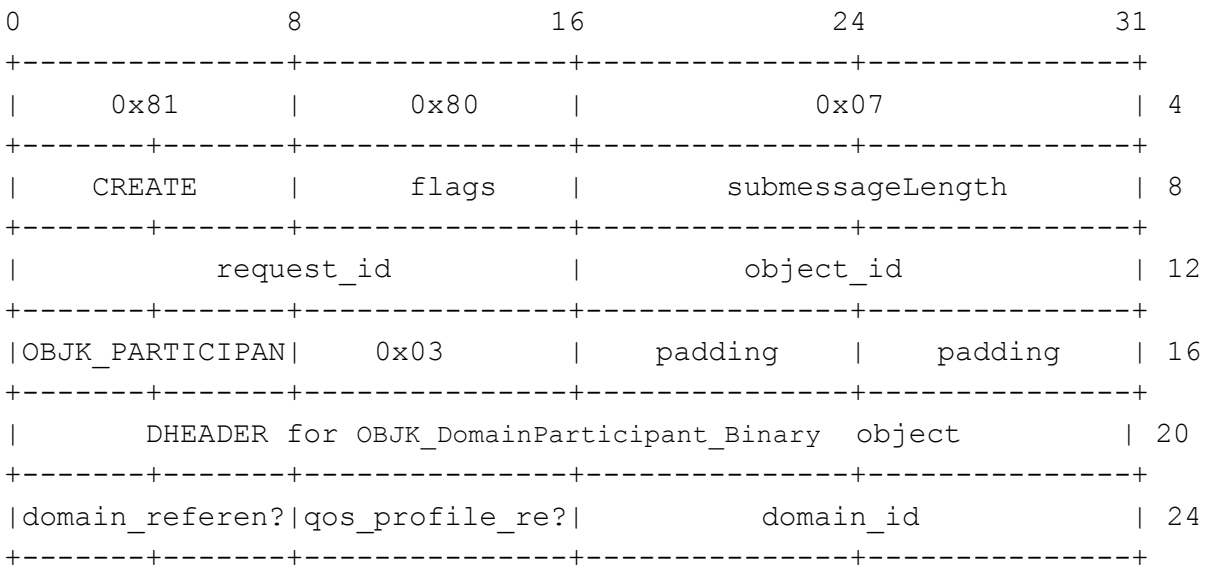


Table 22 describes the bytes in the ***CREATE*** message.

Table 21 Description of the CREATE message for the DomainParticipant using binary representation

Bytes		Description
0-8	Message Header. Same as Table 20.	
4-7	Submessage Header. Similar to Table 20.	
8-23	CREATE_Payload	
	Bytes 8-11 used for BaseObjectRequest (base class of CREATE_Payload). Same as Table 20.	
	Bytes 12-32 used for the ObjectVariant	
	Byte 12	ObjectVariant discriminator = 0x01 Set to OBJK_PARTICIPANT
	Bytes 13-32 are OBJK_Representation3_Base (base class of PARTICIPANT_Representation)	
	Byte 13	OBJK_Representation3_Base discriminator = 0x03 RepresentationFormat set to REPRESENTATION_IN_BINARY
	Bytes 14-15	padding

	Bytes 16-19	DHEADER of OBJK_DomainParticipant_Binary (because extensibility is APPENDABLE) Encodes the endianness and length of the serialized OBJK_DomainParticipant_Binary object Since the length is 2 and the desired endianness is little endian the value of DHEADER is: 0x80000002 = {0x02, 0x00, 0x00, 0x80}
	Byte 20	Optional field domain_reference = 0x00 Set to 0x00 (FALSE) to indicate the field is not present
	Byte 21	Optional field qos_profile_reference = 0x00 Set to 0x00 (FALSE) to indicate the field is not present
	Bytes 22-23 used for the PARTICIPANT_Representation beyond its base class	
	Bytes 22-23	domain_id = {0x00, 0x00} Little endian representation of domain_id 0.

B.2.3. Create a DataWriter using REPRESENTATION_IN_BINARY

The following message would be used by a XRCE Client request a XRCE ProxyClient to create an XRCE DataWriter with ObjectID {0xDD, 0xD5} for topic "Square" using default Qos.

The created XRCE `DataWriter` should belong to an XRCE Publisher with subscriber_id {0xBB, 0xB3}.

The `DataWriter` is represented in binary. Therefore the RepresentationFormat is set to REPRESENTATION_IN_BINARY. In this example it will use little endian encoding.

The binary representation of a DataWriter uses the XCDR serialized representation of the type `OBJK_DataWriter_Binary` defined in Annex A IDL Types as:

```
@extensibility(APPENDABLE)
struct  OBJK_DataWriter_Binary {
    string                                  topic_name;
    @optional OBJK_DataWriter_Binary_Qos  qos;
};
```

The corresponding message is:

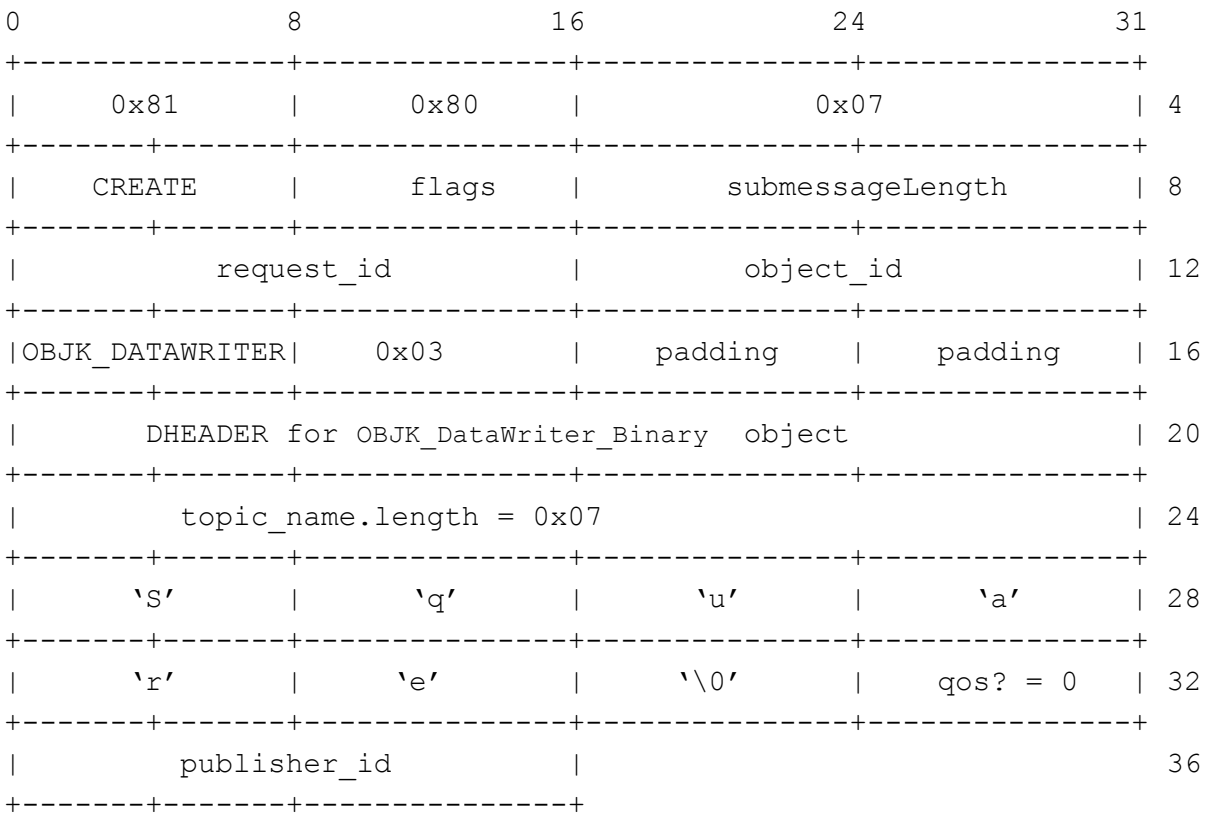


Table 22 describes the bytes in the ***CREATE*** message.

Table 22 Description of the CREATE message for the DataWriter using binary representation and default Qos

Bytes		Description
0-3	Message Header	
	Byte 0	sessionId = 0x81 Indicates session 1 with no client key included in the message.
	Byte 1	streamId=0x80 Selects the builtin reliable stream, see 8.3.2.2
	Bytes 2-3	sequenceNr = 0x07
4-7	Submessage Header	
	Byte 4	submessageId = CREATE = 0x01 See 8.3.5.2
	Byte 5	flags = 0x07 (reuse, replace, little endian)
	Bytes 6-7	submessageLength = 26

		Represented in little endian as {0x1A, 0x00}
8-33	CREATE_Payload	
	Bytes 8-11 used for BaseObjectRequest (base class of CREATE_Payload)	
	Bytes 8-9	BaseObjectRequest request_id = {0xAA , 0x01
	Bytes 10-11	BaseObjectRequest object_id = {0xDD, 0xD5} For a description of the ObjectID see 7.6.
	Bytes 12-32 used for the ObjectVariant	
	Byte 12	ObjectVariant discriminator = 0x05 Set to OBJK_DATAWRITER
	Bytes 13-32 are OBJK_RepresentationBinAndXML_Base (base class of DATAWRITER_Representation)	
	Byte 13	OBJK_RepresentationBinAndXML Base discriminator = 0x03 RepresentationFormat set to REPRESENTATION_IN_BINARY
	Bytes 14-15	padding
	Bytes 16-31 are OBJK_DataWriter_Binary	
	Bytes 16-19	DHEADER of OBJK_DataWriter_Binary (because extensibility is APPENDABLE) Encodes the endianness and length of the serialized OBJK_DataWriter_Binary object Since the length is 12 and the desired endianness is little endian the value of DHEADER is: 0x8000000C encoded in little endian as {0x0C, 0x00, 0x00, 0x80}
	Bytes 20-23	topic_name.length = 0x07 Encodes length of the string represented in little endian as {0x07, 0x00, 0x00, 0x00}
	Bytes 24-30	Characters of the topic_name string, including the terminating NUL. Total of 7 characters
	Byte 31	Optional field qos = 0x00 Set to 0x00 (FALSE) to indicate the qos field is not present
	Bytes 32-33 used for the DATAWRITER_Representation beyond its base class	
	Bytes 32-33	publisher_id = {0xBB, 0xB3}

B.2.4. Create a DataWriter with Qos using REPRESENTATION_IN_BINARY

The following message would be used by a XRCE Client request a XRCE ProxyClient to create an XRCE DataWriter with ObjectID {0xDD, 0xD5} for topic “Square” specifying the Qos in binary.

The created XRCE `DataWriter` should belong to an XRCE Publisher with publisher_id {0xBB, 0xB3}.

The desired DataWriter Qos deviates from the DDS default in that it has RELIABILITY policy set to BEST_EFFORT, HISTORY policy set to KEEP_ALL and DEADLINE policy set to a period of 2 minutes.

The `DataWriter` is represented in binary. Therefore the RepresentationFormat is set to REPRESENTATION_IN_BINARY. In this example it will use little endian encoding.

The binary representation of a DataWriter uses the XCDR serialized representation of the type `OBJK_DataWriter_Binary` defined in Annex A IDL Types as:

```
@extensibility(APPENDABLE)
struct  OBJK_DataWriter_Binary {
    string                                  topic_name;
    @optional OBJK_DataWriter_Binary_Qos  qos;
};
```

Where `OBJK_DataWriter_Binary_Qos` is defined in Annex A IDL Types as:

```
@extensibility(FINAL)
struct  OBJK_Endpoint_Binary_Qos {
    EndpointQosFlags            qos_flags;
    @optional unsigned short    history_depth;
    @optional unsigned long     deadline_msec;
    @optional unsigned long     lifespan_msec;
    @optional sequence<octet>   user_data;
};
@extensibility(FINAL)
struct  OBJK_DataWriter_Binary_Qos :  OBJK_Endpoint_Binary_Qos {
    @optional unsigned long     ownership_strength;
};
```

The corresponding message is:

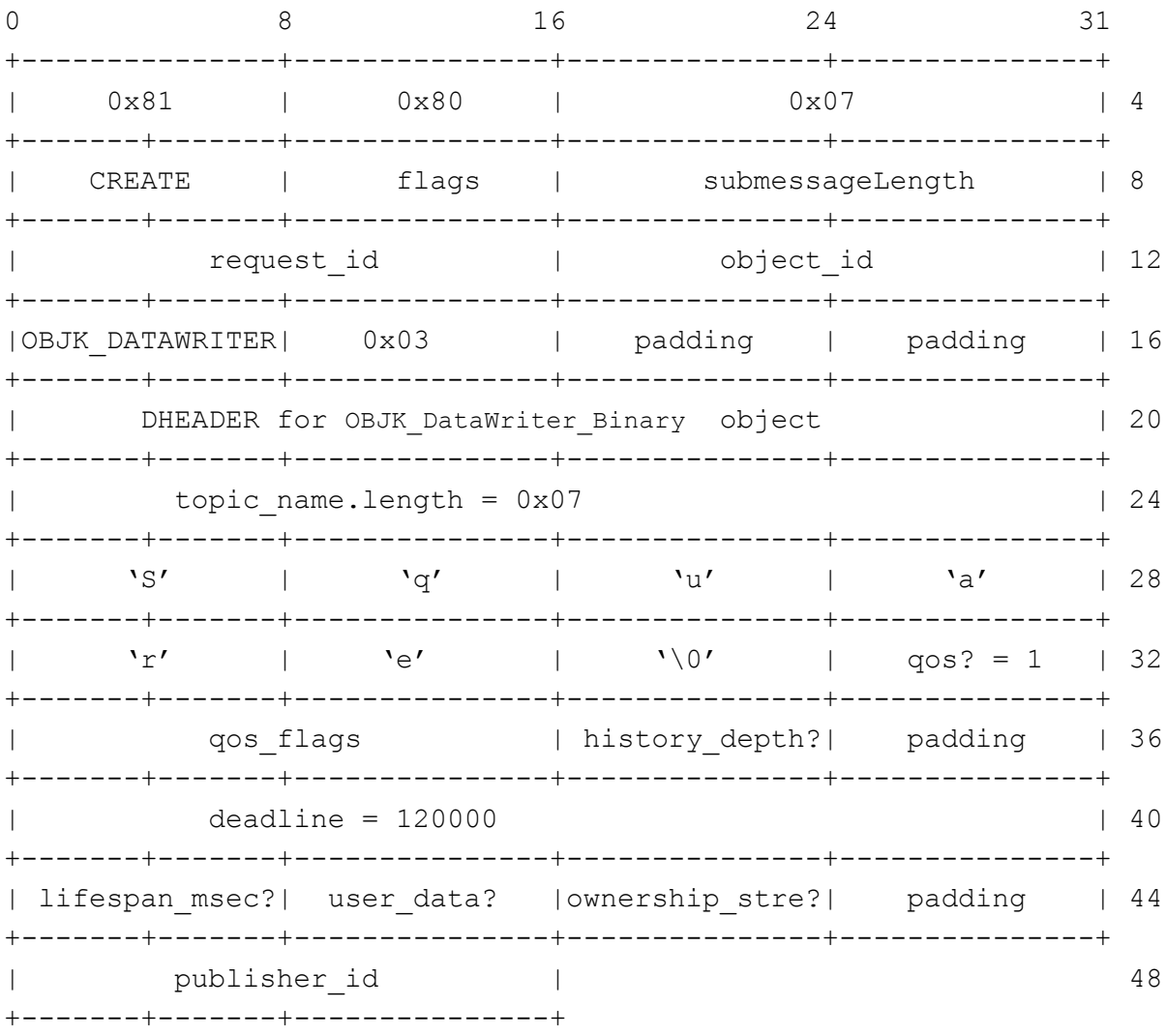


Table 23 describes the bytes in the ***CREATE*** message.

Table 23 Description of the CREATE message for the DataWriter using binary representation and Qos

Bytes		Description
0-8	Message Header. Same as Table 22.	
4-7	Submessage Header. Similar to Table 22.	
8-45	CREATE_Payload	
	Bytes 8-11 used for BaseObjectRequest (base class of CREATE_Payload). Same as Table 22	
	Bytes 12-32 used for the ObjectVariant	
	Byte 12 -30	Same as Table 22

	Byte 31	qos? Set to 0x01 (TRUE) to indicate the qos field is present
	Bytes 32-45: OBJK_Endpoint_Binary_Qos (base class of OBJK_DataWriter_Binary_Qos)	
	Bytes 32-33	qos_flags = 0x0003 Indicates the flags for is_reliable and is_history_keep_all are both set.
	Byte 34	history_depth? Set to 0x00 (FALSE)
	Byte 35	padding
	Bytes 36-39	deadline = 120000 = 0x1D4C0 Period of 2 minutes in milliseconds. In little endian = {0xC0, 0xD4, 0x01, 0x00}
	Byte 40	lifespan? Set to 0x00 (FALSE)
	Byte 41	user_data? Set to 0x00 (FALSE)
	Byte 42	ownership_strength? Set to 0x00 (FALSE)
	Byte 43	padding
	Bytes 44-45	publisher_id = {0xBB, 0xB3}

B.2.5. Create a DataWriter using REPRESENTATION_AS_XML_STRING

The following message would be used by a XRCE `Client` request a XRCE `ProxyClient` to create a `DataWriter` with ObjectID {0xDD, 0xD5}.

The created XRCE `DataWriter` should belong to an XRCE Publisher with publisher_id {0xBB, 0xB3}.

The `DataWriter` is represented in XML. Therefore the `RepresentationFormat` is set to `REPRESENTATION_AS_XML_STRING`.

The XML representation references a Topic “Square” and QosProfile “MyQosLib:MyProfile” both known to the XRCE Agent and uses the XML element:

```
<data_writer name="MyWriter" topic_ref="Square">
    <data_writer_qos base_name="MyQosLib::MyProfile">
        <deadline>
            <period><sec>120</sec></period>
        </deadline>
    </data_writer_qos>
</data_writer>
```

The corresponding message is:

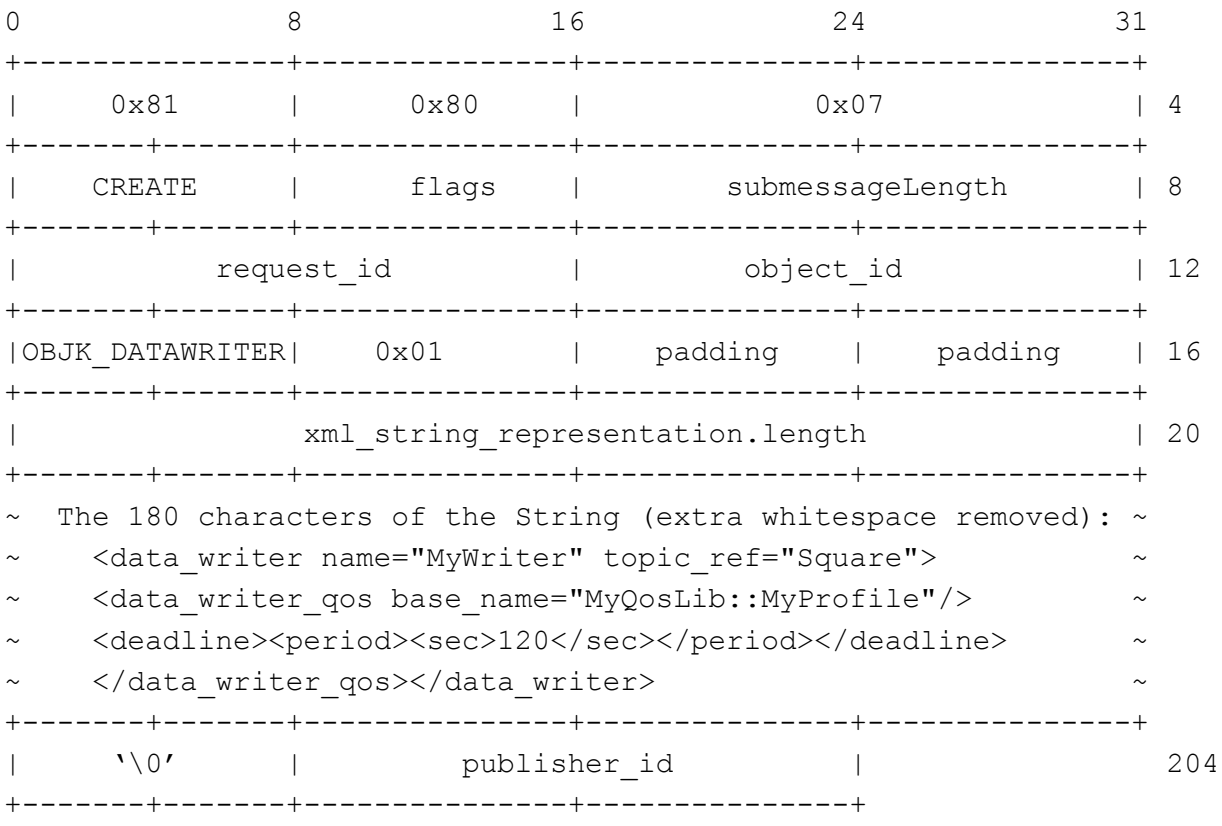


Table 24 describes the bytes in the ***CREATE*** message.

Table 24 Description of the CREATE message for a DataWriter using XML representation

Bytes		Description
0-3	Message Header. Same as Table 22.	
4-7	Submessage Header. Similar to Table 22.	
8-202	CREATE_Payload	
	Bytes 8-11 used for BaseObjectRequest (base class of CREATE_Payload)	
	Bytes 8-9	BaseObjectRequest request_id = {0xAA , 0x01
	Bytes 10-11	BaseObjectRequest object_id = {0xDD, 0xD5} For a description of the ObjectID see 7.6.
	Bytes 12-202 used for the ObjectVariant	
	Byte 12	ObjectVariant discriminator = 0x05 Set to OBJK_DATAWRITER

	Bytes 13-202 are OBJK_RepresentationBinAndXML_Base (base class of DATAWRITER_Representation)	
	Byte 13	OBJK_RepresentationBinAndXML Base discriminator = 0x02 RepresentationFormat set to REPRESENTATION_AS_XML_STRING
	Bytes 14-15	padding
	Bytes 16-19	xml_string_representation.length = 181 = 0x000000B5 Since flags has the Endianness bit set to 1 it is encoded using little endian as {0xB5, 0x00, 0x00, 0x00}
	Bytes 20-200	Characters of the xml_string_representation string, including the terminating NUL. Total of 181 characters
	Bytes 201-202 used for the DATAWRITER_Representation beyond its base class	
	Bytes 201-202	publisher_id = {0xBB, 0xB3}

B.2.6. Create a DataReader using REPRESENTATION_IN_BINARY

The following message would be used by a XRCE Client request a XRCE ProxyClient to create an XRCE DataReader with ObjectID {0xDD, 0xD6} for topic "Square" using default Qos.

The created XRCE DataReader should belong to an XRCE Subscriber with subscriber_id {0xCC, 0xC4}.

The DataReader is represented in binary. Therefore the RepresentationFormat is set to REPRESENTATION_IN_BINARY. In this example it will use little endian encoding.

The binary representation of a DataWriter uses the XCDR serialized representation of the type `OBJK_DataReader_Binary` defined in Annex A IDL Types as:

```
    @extensibility(APPENDABLE)
    struct  OBJK_DataReader_Binary {
        string                                  topic_name;
        @optional OBJK_DataReader_Binary_Qos  qos;
   };
```

The corresponding message is:

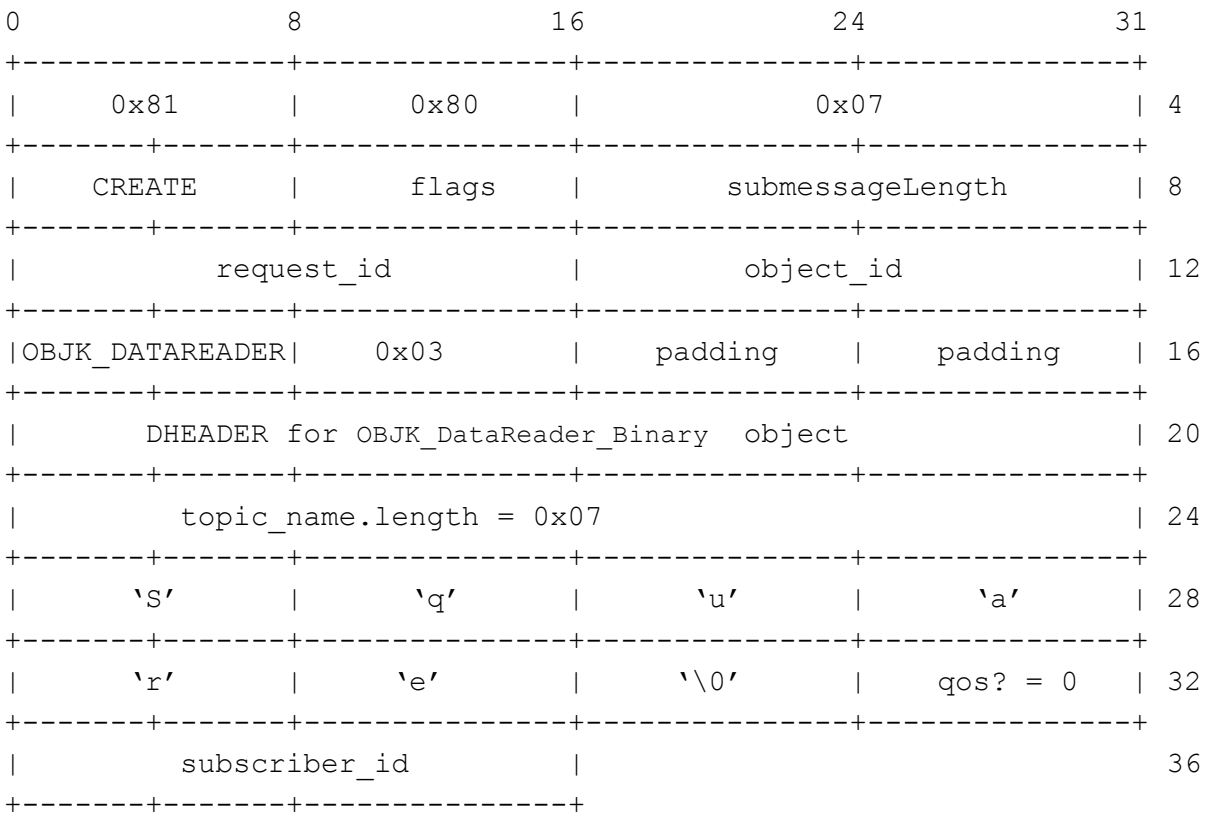


Table 25 describes the bytes in the ***CREATE*** message.

Table 25 Description of the CREATE message for the DataReader using binary representation and default Qos

Bytes		Description
0-3	Message Header	
	Byte 0	sessionId = 0x81 Indicates session 1 with no client key included in the message.
	Byte 1	streamId=0x80 Selects the builtin reliable stream, see 8.3.2.2
	Bytes 2-3	sequenceNr = 0x07
4-7	Submessage Header	
	Byte 4	submessageId = CREATE = 0x01 See 8.3.5.2
	Byte 5	flags = 0x07 (reuse, replace, little endian)
	Bytes 6-7	submessageLength = 26

		Represented in little endian as {0x1A, 0x00}
8-33	CREATE_Payload	
	Bytes 8-11 used for BaseObjectRequest (base class of CREATE_Payload)	
	Bytes 8-9	BaseObjectRequest request_id = {0xAA , 0x01
	Bytes 10-11	BaseObjectRequest object_id = {0xDD, 0xD6} For a description of the ObjectID see 7.6.
	Bytes 12-32 used for the ObjectVariant	
	Byte 12	ObjectVariant discriminator = 0x05 Set to OBJK_DATAREADER
	Bytes 13-32 are OBJK_RepresentationBinAndXML_Base (base class of DATAREADER_Representation)	
	Byte 13	OBJK_RepresentationBinAndXML Base discriminator = 0x03 RepresentationFormat set to REPRESENTATION_IN_BINARY
	Bytes 14-15	padding
	Bytes 16-19	DHEADER of OBJK_DaraReader_Binary (because extensibility is APPENDABLE) Encodes the endianness and length of the serialized OBJK_DaraReader_Binary object Since the length is and the desired endianness is little endian the value of DHEADER is: {0xB5, 0x00, 0x00, 0x00}
	Bytes 24-30	topic_name.length = 0x07 Encodes length of the string represented in little endian as {0x07, 0x00, 0x00, 0x00}
	Bytes 24-30	Characters of the topic_name string, including the terminating NUL. Total of 7 characters
	Byte 31	Optional field qos = 0x00 Set to 0x00 (FALSE) to indicate the qos field is not present
	Bytes 32-33 used for the DATAREADER_Representation beyond its base class	
	Bytes 32-33	subscriber_id = {0xCC, 0xC4}

B.2.7. Create a DataReader with Qos using REPRESENTATION_IN_BINARY

The following message would be used by a XRCE Client request a XRCE ProxyClient to create an XRCE DataReader with ObjectID {0xDD, 0xD6} for topic “Square” specifying the Qos in binary.

The created XRCE DataReader should belong to an XRCE Subscriber with subscriber_id {0xCC, 0xC4}.

The desired DataReader Qos deviates from the DDS default in that it has HISTORY policy set to KEEP_ALL and DEADLINE policy set to a period of 5 minutes.

In addition the DataReader installs a filter with the expression “x>100”.

The DataReader is represented in binary. Therefore the RepresentationFormat is set to REPRESENTATION_IN_BINARY. In this example it will use little endian encoding.

The binary representation of a DataWriter uses the XCDR serialized representation of the type `OBJK_DataReader_Binary` defined in Annex A IDL Types as:

```
@extensibility(APPENDABLE)
struct  OBJK_DataReader_Binary {
    string                                topic_name;
    @optional OBJK_DataReader_Binary_Qos  qos;
};
```

Where `OBJK_DataReader_Binary_Qos` is defined in Annex A IDL Types as:

```
@extensibility(FINAL)
struct  OBJK_Endpoint_Binary_Qos {
    EndpointQosFlags            qos_flags;
    @optional unsigned short    history_depth;
    @optional unsigned long     deadline_msec;
    @optional unsigned long     lifespan_msec;
    @optional sequence<octet>   user_data;
};
@extensibility(FINAL)
struct  OBJK_DataReader_Binary_Qos :  OBJK_Endpoint_Binary_Qos {
  @optional unsigned long     timebasedfilter_msec;
  @optional string            contentbased_filter;
};
```

The corresponding message is:

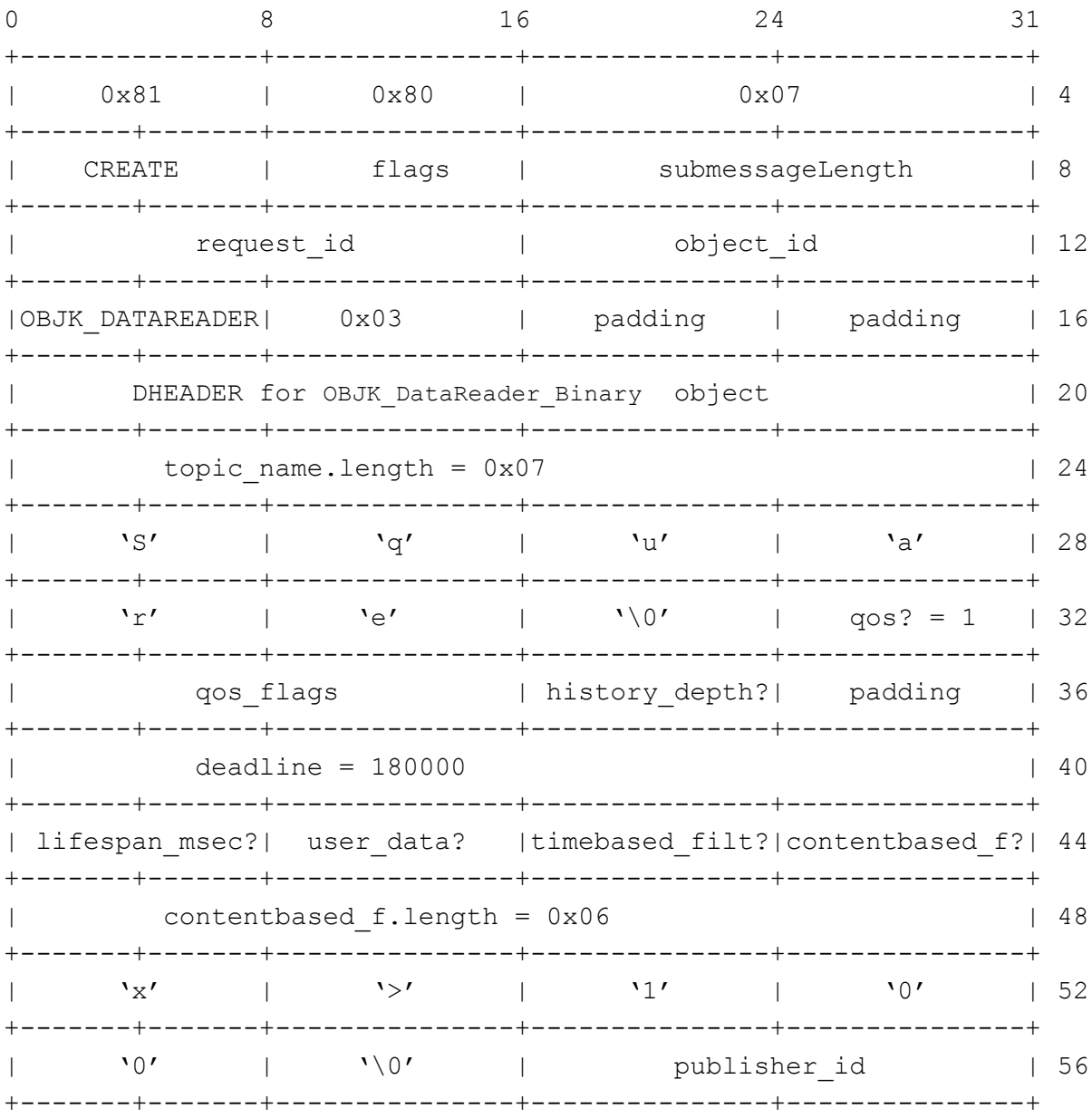


Table 26 describes the bytes in the ***CREATE*** message.

Table 26 Description of the CREATE message for the DataWriter using binary representation and Qos

Bytes		Description
0-8	Message Header. Same as Table 25.	
4-7	Submessage Header. Similar to Table 25.	
8-55	CREATE_Payload	
	Bytes 8-11 used for BaseObjectRequest (base class of CREATE_Payload). Same as Table 25	

	Bytes 12-55 used for the ObjectVariant	
	Byte 12 -30	Same as Table 25
	Byte 31	qos? Set to 0x01 (TRUE) to indicate the qos field is present
	Bytes 32-43: OBJK_Endpoint_Binary_Qos (base class of OBJK_DataWriter_Binary_Qos)	
	Bytes 32-33	qos_flags = 0x0002 Only the flags for is_history_keep_all is set.
	Byte 34	history_depth? Set to 0x00 (FALSE)
	Byte 35	padding
	Bytes 36-39	deadline = 180000 = 0x2BF20 Period of 3 minutes in milliseconds. In little endian = {0x20, 0xBF, 0x02, 0x00}
	Byte 40	lifespan? Set to 0x00 (FALSE)
	Byte 41	user_data? Set to 0x00 (FALSE)
	Bytes 42-53: OBJK_DataReader_Binary_Qos beyond OBJK_Endpoint_Binary_Qos	
	Byte 42	timebased_filter? Set to 0x00 (FALSE)
	Byte 43	contentbased_filter? Set to 0x01 (TRUE)
	Byte 44-47	contentbased_filter.length = 0x06 Encodes length of the string represented in little endian as {0x06, 0x00, 0x00, 0x00}
	Byte 48-53	Characters of the contentbased_filter string, including the terminating NUL. Total of 6 characters
	Bytes 54-55: CREATE_Payload beyond BaseObjectRequest	
	Bytes 54-55	publisher_id = {0xBB, 0xB3}

B.3. WRITE_DATA message examples

B.3.1. Writing a single data sample

The following message could be used by a XRCE Client to write data using an already created XRCE DataWriter, identified by object_id {0x44, 0x05}. It uses an existing session with session_id 0xDD to send the request.

The XCREClient uses request_id = {0xAA, 0x01} to identify this request.

The XRCE Client writes a single sample of data with no meta-data. See 7.7.1 and 7.7.2 for a description of the different formats available to write and read data. Therefore the payload of the WRITE_DATA message is the XCDR serialized representation of the WRITE_DATA_Payload_Data type defined in Annex A IDL Types.

```
@extensibility(FINAL)
struct SampleData {
    XCDRSerializedBuffer serialized_data;
};

@extensibility(FINAL)
struct WRITE_DATA_Payload_Data : BaseObjectRequest {
    SampleData              data;
};
```

In this example we assume the data written corresponds to a struct `Temparature` type described in the following IDL:

```
@extensibility(FINAL)
struct Temperature {
    short  value;
};
```

Furthermore we assume that the value written is 25.

The corresponding message is:

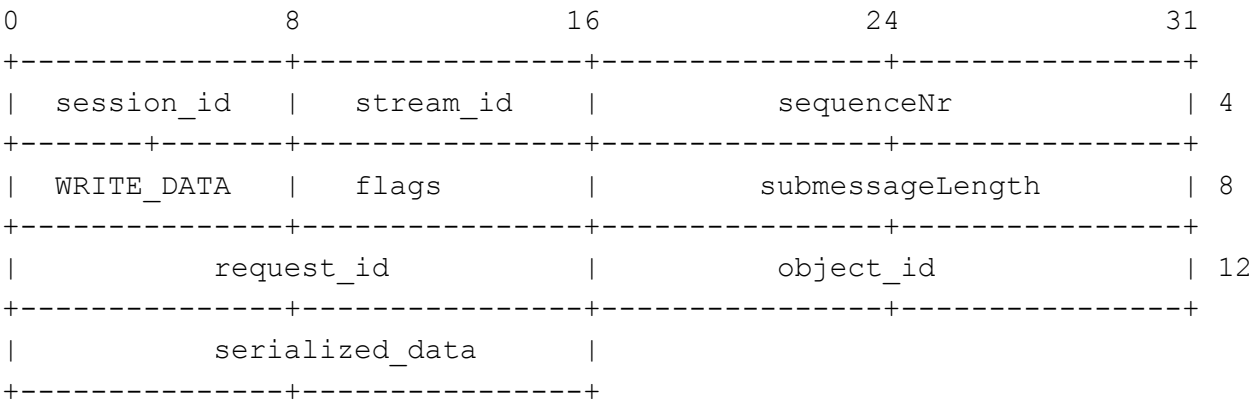


Table 27 describes each of the bytes in the message.

Table 27 Description of the READ_DATA (single sample) example bytes

Bytes		Description
0-3	Message Header	

	Byte 0	sessionId = 0xDD
	Byte 1	streamId=0x80 Selects STREAMID_BUILTIN_RELIABLE, see 8.3.2.2
	Bytes 2-3	sequenceNr = 1 Represented in little endian (see 8.3.2.3) as {0x01, 0x00}
4-7	Submessage Header	
	Byte 4	submessageId = WRITE_DATA = 0x07
	Byte 5	flags = 0x01 Bit 0 (lowest bit) = 1 indicate little endian encoding Bits 1, 2, 3 set to zero indicate payload DataFormat is FORMAT_DATA. See 8.3.5.8.1.
	Bytes 6-7	submessageLength = 6 = 0x0006 Represented in little endian (see 8.3.4.3) as {0x06, 0x00}
8-13	WRITE_DATA_Payload	
	Bytes 8-11 used for BaseObjectRequest (base class of WRITE_DATA_Payload)	
	Bytes 8-9	request_id = {0xAA , 0x01}
	Bytes 10-11	object_id = {0x44, 0x05}
	Bytes 12-13 are used for SampleData (remaining of WRITE_DATA_Payload after base class)	
	Byte 12-13	serialized_data = {0x19, 0x00} Little endian serialized representation of the Temperature value of 25 (in hex 0x0019).

B.3.2. Writing a sequence of data samples with no sample information

The following message could be used by a XRCE Client to write data using an already created XRCE DataWriter, identified by object_id {0x44, 0x05}. It uses an existing session with session_id 0xDD to send the request.

The XCREClient uses request_id = {0xAA, 0x01} to identify this request.

The XRCE Client writes a sequence of bare data samples with no meta-data. See 7.7.1 and 7.7.2 for a description of the different formats available to write and read data. Therefore the payload of the WRITE_DATA message is the XCDR serialized representation of the `WRITE_DATA_Payload_DataSeq` type defined in Annex A IDL Types.

```
@extensibility(FINAL)
```

```
struct SampleData {
    XCDRSerializedBuffer serialized_data;
};

@extensibility(FINAL)
struct WRITE_DATA_Payload_DataSeq : BaseObjectRequest {
    sequence<SampleData>    data_seq;
};
```

In this example we assume the data written corresponds to a two values of the struct `Temparature` type described in the following IDL:

```
@extensibility(FINAL)
struct Temperature {
    short  value;
};
```

Furthermore we assume that there are five values written: 20, 17, 26, and 40 .

The corresponding message is:

```
0               8               16               24               31
+---------------+---------------+----------------+----------------+
|  session_id   |   stream_id   |          sequenceNr             | 4
+-------+-------+---------------+----------------+----------------+
|  WRITE_DATA   |   flags       |         submessageLength        | 8
+---------------+---------------+----------------+----------------+
|           request_id          |          object_id              | 12
+---------------+---------------+----------------+----------------+
|                    data_seq.length = 4                          | 16
+---------------+---------------+----------------+----------------+
|  data_seq[0].serialized_data  |  data_seq[1].serialized_data    | 20
+---------------+---------------+----------------+----------------+
|  data_seq[2].serialized_data  |  data_seq[3].serialized_data    | 24
+---------------+---------------+----------------+----------------+
```

Table 28 describes each of the bytes in the message.

Table 28 Description of the READ_DATA (single sample) example bytes

Bytes		Description
0-3	Submessage Header similar to Table 27	

4-7	Submessage Header	
	Byte 4	submessageId = WRITE_DATA = 0x07
	Byte 5	flags = 0x09 Bit 0 (lowest bit) = 1 indicate little endian encoding Bits 3, 2, 1 respectively set to 1, 0, 0, indicate payload DataFormat is FORMAT_DATA_SEQ See 8.3.5.8.1.
	Bytes 6-7	submessageLength = 6 = 0x0006 Represented in little endian (see 8.3.4.3) as {0x06, 0x00}
8-23	WRITE_DATA_Payload	
	Bytes 8-11 used for BaseObjectRequest (base class of WRITE_DATA_Payload) Same as Table 27	
	Bytes 12-13 are used for SampleData (remaining of WRITE_DATA_Payload after base class)	
	Bytes 12-15	data_seq.length = 4, Encoded in little endian as {0x04, 0x00, 0x00, 0x00}
	Bytes 16-23	Little endian serialized representation of the 4 short temperature values 20, 17, 26, and 40: {{0x14, 0x00}, {0x11, 0x00}, {0x1A, 0x00} {0x24, 0x00}}

B.3.3. Writing a single data sample with timestamp metadata

The following message could be used by a XRCE Client to write data using an already created XRCE DataWriter, identified by object_id {0x44, 0x05}. It uses an existing session with session_id 0xDD to send the request.

The XCREClient uses request_id = {0xAA, 0x01} to identify this request.

The XRCE Client writes a single sample of data with additional metadata allowing it to put a timestamp and also notify of instance lifecycle changes such as the deletion of an instance. See 7.7.1 and 7.7.2 for a description of the different formats available to write and read data.

The payload of the WRITE_DATA message is the XCDR serialized representation of the `WRITE_DATA_Payload_Sample` type defined in Annex A IDL Types.

```
@bit_bound(8)
bitmask SampleInfoFlags {
    @position(0) INSTANCE_STATE_UNREGISTERED,
    @position(1) INSTANCE_STATE DISPOSED,
    @position(2) VIEW_STATE_NEW,
    @position(3) SAMPLE_STATE_READ,
};
```

```
@extensibility(FINAL)
struct SeqNumberAndTimestamp {
    unsigned long   sequence_number;
    unsigned long   session_time_offset; // milliseconds up to 53 days
};

@extensibility(FINAL)
union SampleInfoDetail switch(SampleInfoFormat) {
  case FORMAT_EMPTY:
  case FORMAT_SEQNUM:
    unsigned long   sequence_number;
  case FORMAT_TIMESTAMP:
    unsigned long   session_time_offset; // milliseconds up to 53 days
  case FORMAT_TIMESTAMP:
    SeqNumberAndTimestamp seqnum_n_timestamp;
};

@extensibility(FINAL)
struct SampleInfo {
    SampleInfoFlags  state; //Combines SampleState, InstanceState, ViewState
    SampleInfoDetail detail;
};

@extensibility(FINAL)
struct SampleData {
    XCDRSerializedBuffer serialized_data;
};
@extensibility(FINAL)
struct Sample {
  SampleInfo   info;
  SampleData   data;
};

@extensibility(FINAL)
struct WRITE_DATA_Payload_Sample : BaseObjectRequest {
    Sample                sample;
};
```

In this example we assume the data written corresponds to a struct `Temparature` type described in the following IDL:

```
@extensibility(FINAL)
struct Temperature {
    short  value;
};
```

Furthermore we assume that the value written is 25.

The corresponding message is:

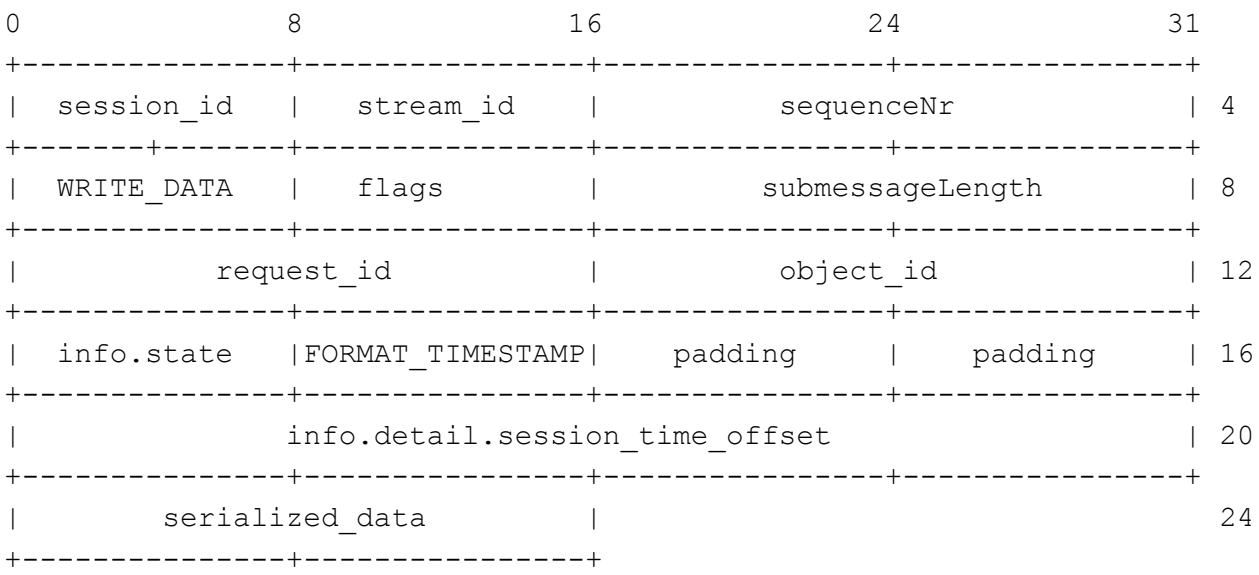


Table 29 describes each of the bytes in the message.

Table 29 Description of the READ_DATA (single sample) example bytes

Bytes		Description
0-3	Submessage Header similar to Table 27	
4-7	Submessage Header	
	Byte 4	submessageId = WRITE_DATA = 0x07
	Byte 5	flags = 0x03 Bit 0 (lowest bit) = 1 indicate little endian encoding Bits 3, 2, 1 respectively set to 0, 0, 1, indicate payload DataFormat is FORMAT_Sample. See 8.3.5.8.1.
	Bytes 6-7	submessageLength = 13 = 0x000D Represented in little endian (see 8.3.4.3) as {0x06, 0x00}

8-21	WRITE_DATA_Payload	
	Bytes 8-11 used for BaseObjectRequest (base class of WRITE_DATA_Payload) Same as Table 27	
	Bytes 12-21 are used for Sample (remaining of WRITE_DATA_Payload after base class)	
	Byte 12	info.state = 0x00 The state bits indicate the instance is ALIVE (the flags for unregistered and disposed are both zero).
	Byte 13	info.detail.discriminator = FORMAT_TIMESTAMP
	Bytes 14-15	padding
	Bytes 16-19	info. Detail.session_time_offset
	Bytes 20-21	serialized_data. Little endian serialized representation of the short temperature value 25: {0x19, 0x00}

B.3.4. Writing a disposed data sample

The following message could be used by a XRCE Client to write data using an already created XRCE DataWriter, identified by object_id {0x44, 0x05}. It uses an existing session with session_id 0xDD to send the request.

The XCREClient uses request_id = {0xAA, 0x01} to identify this request.

The XRCE Client writes a single sample of data with additional metadata allowing it to put a timestamp and also notify of instance lifecycle changes such as the deletion of an instance. See 7.7.1 and 7.7.2 for a description of the different formats available to write and read data.

The payload of the WRITE_DATA message is the XCDR serialized representation of the `WRITE_DATA_Payload_Sample` type defined in Annex A IDL Types. See also B.3.3 for the types used in this message.

In this example we assume the data written corresponds to a keyed data-type. The structure `TemparatureSensor` described in the following IDL:

```
@extensibility(FINAL)
struct TemperatureSensor {
    @key   octet sensor_id[4];
    short  sensor_value;
};
```

Furthermore the example assumes that the written data has sensor_id = {0x11, 0x22, 0x33, 0x64} and sensor_value = 25.

The corresponding message is:

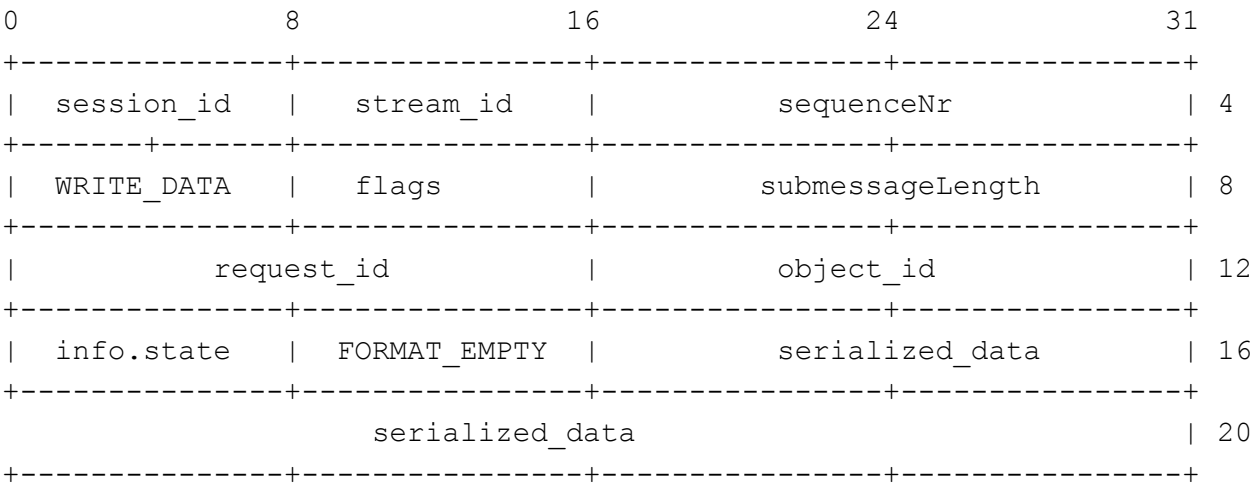


Table 30 describes each of the bytes in the message.

Table 30 Description of the READ_DATA (single sample) example bytes

Bytes		Description
0-3	Submessage Header similar to Table 27	
4-7	Submessage Header	
	Byte 4	submessageId = WRITE_DATA = 0x07
	Byte 5	flags = 0x03 Bit 0 (lowest bit) = 1 indicate little endian encoding Bits 3, 2, 1 respectively set to 0, 0, 1, indicate payload DataFormat is FORMAT_Sample. See 8.3.5.8.1.
	Bytes 6-7	submessageLength = 6 = 0x0006 Represented in little endian (see 8.3.4.3) as {0x06, 0x00}
8-19	WRITE_DATA_Payload	
	Bytes 8-11 used for BaseObjectRequest (base class of WRITE_DATA_Payload) Same as Table 27	
	Bytes 12-19 are used for Sample (remaining of WRITE_DATA_Payload after base class)	
	Byte 12	info.state = 0x02 The state bits indicate the instance is DISPOSED (the flag for unregistered is zero but the flag for disposed is one).

	Byte 13	Info.detail.discriminator = FORMAT_NONE Indicates no additional information beyond the state.
	Bytes 14-19	Serialized_data = { {0x11, 0x22, 0x33, 0x64}, {0x19, 0x00}} Little endian serialized representation of the sensor data. First four bytes are the sender_id and following two bytes the sensor_value.

B.4. READ_DATA message examples

B.4.1. Reading a single data sample

The following message could be used by a XRCE Client to read data from an already created XRCE DataReader, identified by object_id {0x44, 0x06}. It uses an (already created) session with session_id 0xDD to send the request.

The XCREClient uses request_id = {0xAA, 0x01} to identify this request.

The ReadSpecification does not specify a content filter and requests a single data sample with no sample information.

The payload of the READ_DATA message is the XCDR serialized representation of the READ_DATA_Payload type defined in Annex A IDL Types.

```
@extensibility(APPENDABLE)
struct DataDeliveryControl {
    unsigned short max_samples;
    unsigned short max_elapsed_time;
    unsigned short max_bytes_per_second;
    unsigned short min_pace_period;  // milliseconds
};

@extensibility(FINAL)
struct ReadSpecification  {
    DataFormat                  data_format;
    @optional string            content_filter_expression;
    @optional DataDeliveryControl delivery_control;
};

@extensibility(FINAL)
struct READ_DATA_Payload : BaseObjectRequest {
    ReadSpecification               read_specification;
};
```

The corresponding message is:

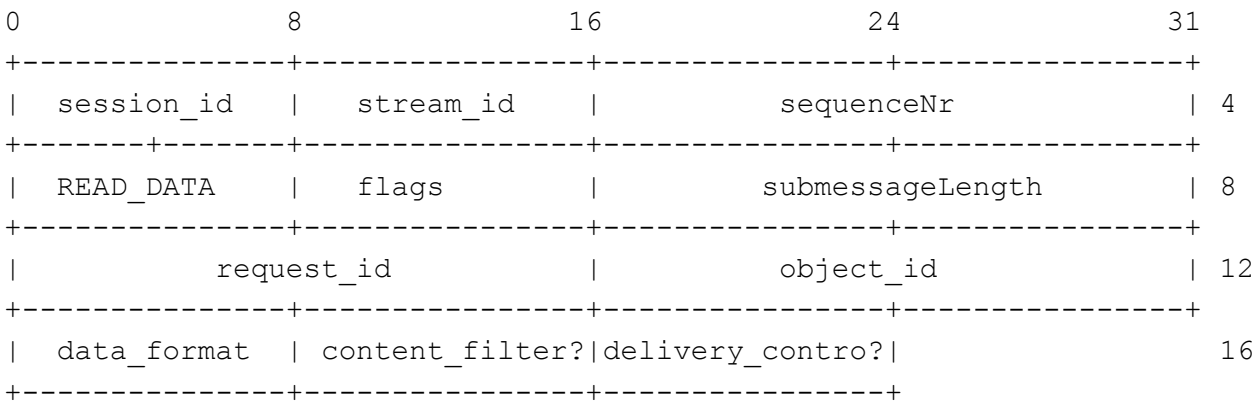


Table 31 describes each of the bytes in the message.

Table 31 Description of the READ_DATA (single sample) example bytes

Bytes		Description
0-3	Message Header	
	Byte 0	sessionId = 0xDD
	Byte 1	streamId=0x80 Selects STREAMID_BUILTIN_RELIABLE, see 8.3.2.2
	Bytes 2-3	sequenceNr = 1 Represented in little endian (see 8.3.2.3) as {0x01, 0x00}
4-7	Submessage Header	
	Byte 4	submessageId = READ_DATA = 0x07
	Byte 5	flags = 0x01 (little endian)
	Bytes 6-7	submessageLength = 7= 0x0007 Represented in little endian (see 8.3.4.3) as {0x07, 0x00}
8-14	READ_DATA_Payload	
	Bytes 8-11 used for BaseObjectRequest (base class of WRITE_DATA_Payload)	
	Bytes 8-9	request_id = {0xAA , 0x01}
	Bytes 10-11	object_id = {0x44, 0x06}

	Bytes 12-14 are used for remaining of READ_DATA_Payload after base class	
	Bytes 12-14 are used for the read_specification of type ReadSpecification	
	Byte 12	read_specification.data_format = 0x00. Encodes the desired DataFormat. In this case selects FORMAT_DATA .
	Byte 13	content_filter_expression? = 0x00. Encodes whether the optional member content_filter_expression is present. In this case it is set to FALSE indicating there it is not present.
	Byte 14	read_specification.delivery_control? = 0x00 Encodes whether the optional member delivery_control is present. In this case it is set to FALSE indicating there is no DataDeliveryControl.

B.4.2. Reading a sequence of data samples with a content filter

The following message could be used by a XRCE Client to request the streaming of data from an already created XRCE DataReader, identified by object_id {0x44, 0x06}. It uses an (already created) session with session_id 0xDD to send the request.

The XCREClient uses request_id = {0xAA, 0x01} to identify this request.

The ReadSpecification requests a stream of no more than 100 data samples, over a time window not to exceed 30 seconds with bandwidth not to exceed 1024 bytes per second and a minimum pace of 1000 milliseconds. It requests samples only with no associated sample information.

In addition the Client request data that matches the content filter expression “x>100”.

This message uses the same data types as B.4.1. The difference is that it selects the DataFormat FORMAT_DATA_SEQ, the ***read_specification*** contains a ***content filter expression*** and a `DataDeliveryControl`.

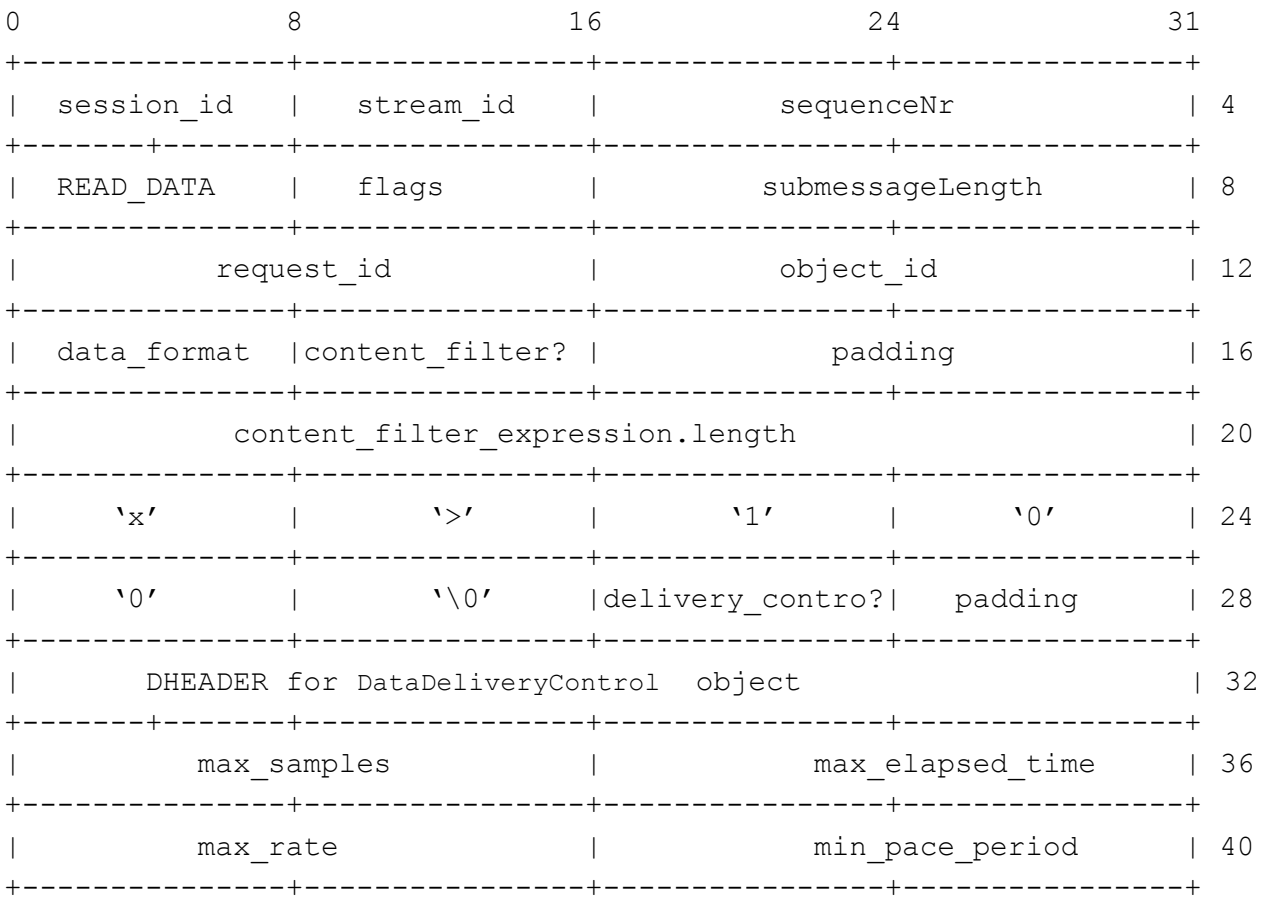


Table 32 describes each of the bytes in the message.

Table 32 Description of the READ_DATA (multiple samples) example bytes

Bytes		Description
0-3	Message Header. Same as Table 31.	
4-7	Submessage Header. Similar to Table 31.	
8-36	READ_DATA_Payload	
	Bytes 8-11 used for BaseObjectRequest (base class of WRITE_DATA_Payload) Same as Table 31.	
	Bytes 12-36 are used for remaining of READ_DATA_Payload after base class	
	Bytes 12-36 are used for the read_specification of type ReadSpecification	
	Byte 12	read_specification.data_format = 0x08 Encodes the desired DataFormat. In this case selects FORMAT_DATA_SEQ.

	Byte 13-25 is used for the content filter expression	
	Byte 13	content_filter_expression? = 0x01. Encodes whether the optional member content_filter_expression is present. In this case it is set to FALSE indicating there it is present.
	Bytes 14-15	padding
	Bytes 16-19	content_filter_expression .length = 6 = 0x00000006 Length of the content_filter_expression string in little endian {0x06,0x00,0x00,0x00}.
	Byte 20-25	Characters of content filter expression, including terminating NUL character.
	Bytes 26-35 are used for the delivery_control of type DataDeliveryControl	
	Byte 26	read_specification.delivery_control? = 0x01 Encodes whether the optional member delivery_control is present. In this case it is set to FALSE indicating there is no DataDeliveryControl.
	Bytes 27	padding
	Bytes 28-31	DHEADER of DataDeliveryControl (because extensibility is APPENDABLE) Encodes the endianness and length of the serialized DataDeliveryControl object Since the length is 8 and the desired endianness is little endian the value of DHEADER is: 0x80000008 = {0x08, 0x00, 0x00, 0x80}
	Byte 28-29	max_samples = 100 = 0x64 Represented in little endian (see flags) as {0x64, 0x00,}
	Byte 30-31	max_elapsed_time = 30000 = 0x7530. Represented in little endian (see flags) as {0x30, 0x75 }
	Byte 32-33	max_rate = 1024 = 0x0400 Represented in little endian (see flags) as {0x00, 0x40}
	Byte 34-35	min_pace_period = 1000 = 0x03E8 Represented in little endian (see flags) as {0xE8, 0x03}

B.5. DATA message examples

B.5.1. Receiving a single data sample

The following message could be used by a XRCE Agent to send a single sample in response to a READ_DATA request from a XRCE Client that used DataFormat FORMAT_DATA.

The example illustrates the response to the request_id {0xAA, 0x01} from the XRCE DataReader with object_id {0x44, 0x06}. It uses the (already created) session with session_id 0xDD to send the data.

The data is sent using best-effort using the builtin stream identified by stream_id STREAMID_BUILTIN_BEST_EFFORTS.

This example also assumes the data being sent corresponds to an object ***foo*** of type `FooType` defined in the IDL below. In the example we assume foo.count is set to 19.

```
@extensibility(FINAL)
struct FooType {
    long count;
};
```

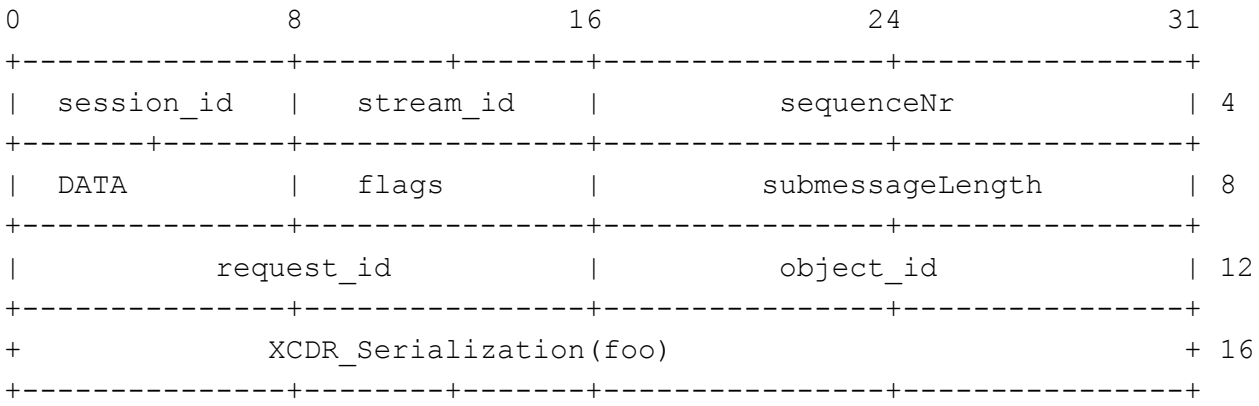


Table 33 Description of the DATA (single samples) example bytes

Bytes		Description
0-3	Message Header	
	Byte 0	sessionId = 0xDD
	Byte 1	streamId=0x01 Selects STREAMID_BUILTIN_BEST_EFFORTS, see 8.3.2.2
	Bytes 2-3	sequenceNr = 1 Represented in little endian (see 8.3.2.3) as {0x01, 0x00}
4-7	Submessage Header	

	Byte 4	submessageId = DATA = 0x09
	Byte 5	flags = 0x00 (big endian)
	Bytes 6-7	submessageLength = 8 = 0x0008 Represented in little endian (see 8.3.4.3) as {0x08, 0x00}
8-15	DATA_Payload_Data (DataFormat was FORMAT_DATA)	
	Bytes 8-9	request_id = {0xAA , 0x01}
	Bytes 10-11	object_id = {0x44, 0x06}
	Byte 12-15	XCDR Serialization of foo of type FooType. Flags is 0x00 so the representation is Big Endian. The resulting for foo.count = 19 is {0x00, 0x00, 0x00, 0x13}.

B.5.2. Receiving a sequence of samples without SampleInfo

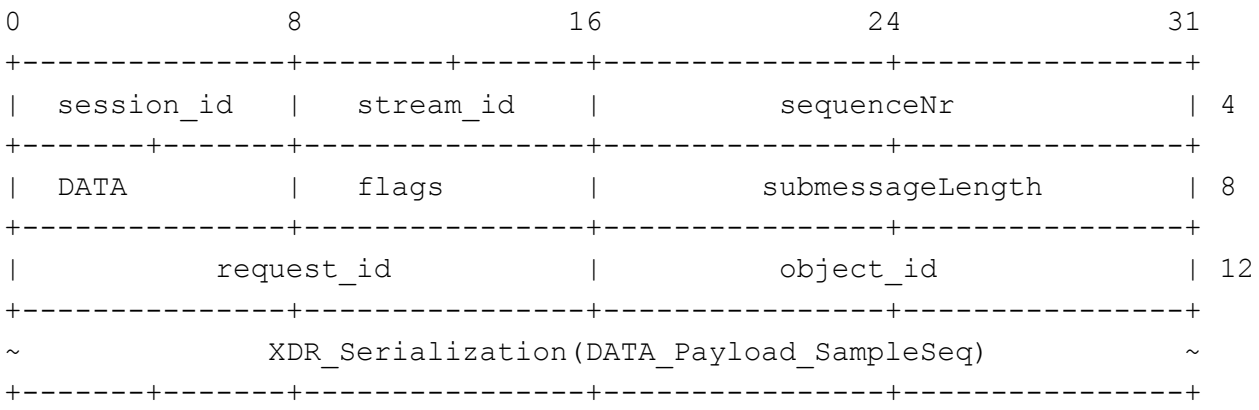
The following message could be used by a XRCE Agent to send a sequence of samples in response to a READ_DATA request from a XRCE Client that used DataFormat FORMAT_DATA_SEQ.

The example illustrates the response to the request_id {0xAA, 0x02} from the XRCE DataReader with object_id {0x44, 0x06}. It uses the (already created) session with session_id 0xDD to send the data.

The data is sent using best-effort using the builtin stream identified by stream_id STREAMID_BUILTIN_BEST_EFFORTS.

This example also assumes the data being sent corresponds to a sequence of two objects ***foo1*** and ***foo1*** of type `FooType` defined in the IDL below In the example we assume foo1.count is set to 1 and foo2.count is set to 1.

```
@extensibility(FINAL)
struct FooType {
    long count;
};
```



The serialization of DATA_Payload_SampleSeq can be expanded as:

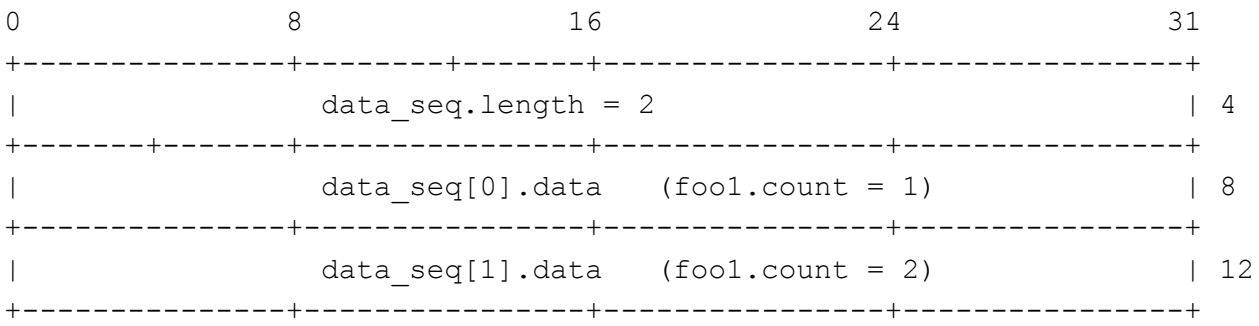


Table 34 Description of the DATA (sample sequence) example bytes

Bytes		Description
0-3	Message Header	
	Byte 0	sessionId = 0xDD
	Byte 1	streamId=0x01 Selects STREAMID_BUILTIN_BEST_EFFORTS, see 8.3.2.2
	Bytes 2-3	sequenceNr = 1 Represented in little endian (see 8.3.2.3) as {0x0A, 0x00}
4-7	Submessage Header	
	Byte 4	submessageId = DATA = 0x08
	Byte 5	flags = 0x00 (big endian)
	Bytes 6-7	submessageLength = 16 = 0x0010 Represented in little endian (see 8.3.4.3) as {0x10, 0x00}

8-23	DATA_Payload_DataSeq (DataFormat was FORMAT_DATA_SEQ)	
	Bytes 8-9	request_id = {0xAA , 0x01}
	Bytes 10-11	object_id = {0x44, 0x06}
	Bytes 12-15	data_seq.length = 2
	Bytes 16-19	data_seq[0].data
	Bytes 20-23	data_seq[1].data

B.5.3. Receiving a single sample that includes SampleInfo

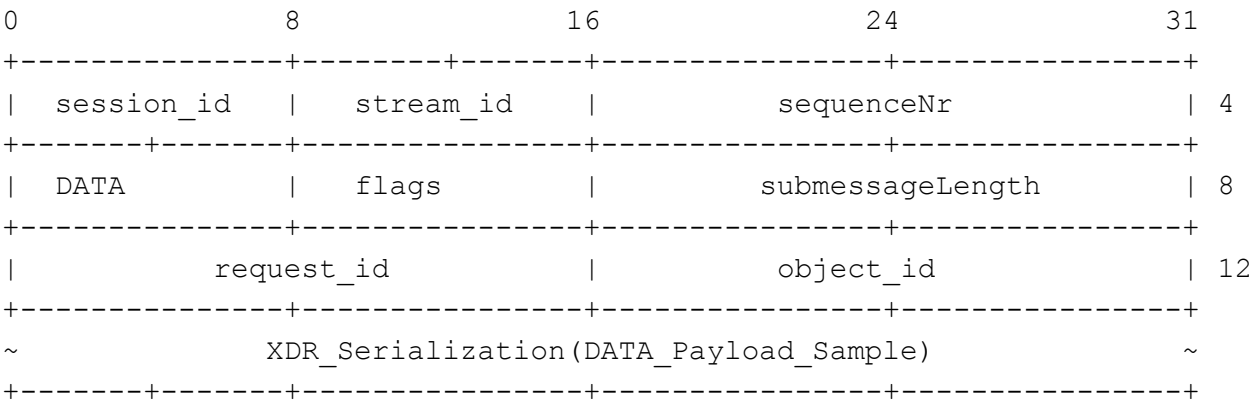
The following message could be used by a XRCE Agent to send a sequence of samples in response to a READ_DATA request from a XRCE Client that used DataFormat FORMAT_SAMPLE.

The example illustrates the response to the request_id {0xAA, 0x02} from the XRCE DataReader with object_id {0x44, 0x06}. It uses the (already created) session with session_id 0xDD to send the data.

The data is sent using best-effort using the builtin stream identified by stream_id STREAMID_BUILTIN_BEST_EFFORTS.

This example also assumes the data being sent corresponds to a sequence of two objects ***foo1*** and ***foo1*** of type `FooType` defined in the IDL below In the example we assume foo1.count is set to 1 and foo2.count is set to 1.

```
@extensibility(FINAL)
struct FooType {
    long count;
};
```



The serialization of DATA_Payload_Sample can be expanded as:

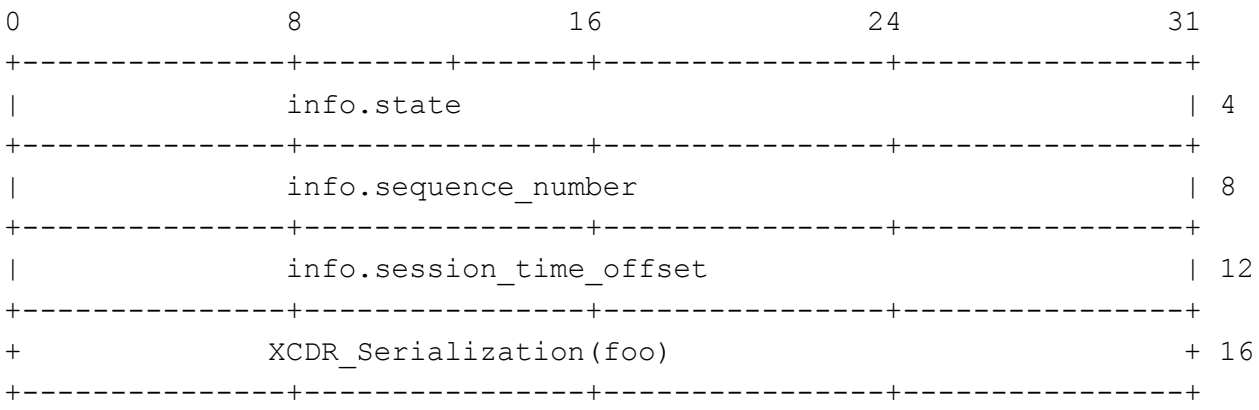
```
@extensibility(FINAL)
union SampleInfoDetail switch(SampleInfoFormat) {
```

```
  case FORMAT_EMPTY:
  case FORMAT_SEQNUM:
    unsigned long   sequence_number;
  case FORMAT_TIMESTAMP:
    unsigned long   session_time_offset; // milliseconds up to 53 days
  case FORMAT_TIMESTAMP:
    SeqNumberAndTimestamp seqnum_n_timestamp;
};

@bit_bound(8)
bitmask SampleInfoFlags {
    @position(0) INSTANCE_STATE_UNREGISTERED,
    @position(1) INSTANCE_STATE DISPOSED,
    @position(2) VIEW_STATE_NEW,
    @position(3) SAMPLE_STATE_READ,
};

@extensibility(FINAL)
struct SampleInfo {
    SampleInfoFlags  state; //Combines SampleState, InstanceState, ViewState
    SampleInfoDetail detail;
};

@extensibility(FINAL)
struct Sample {
  SampleInfo   info;
  SampleData   data;
};
```



B.5.4. Receiving a sequence of samples that includes SampleInfo

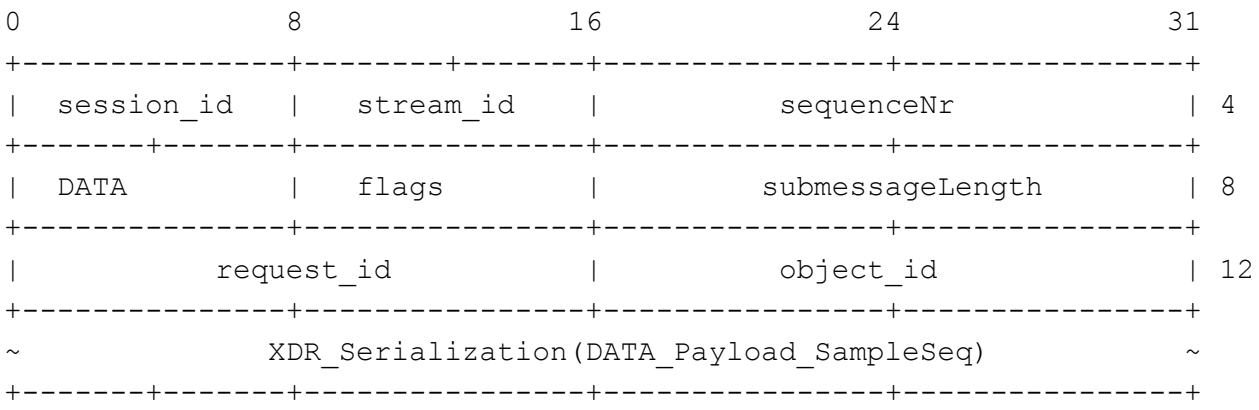
The following message could be used by a XRCE Agent to send a sequence of samples in response to a READ_DATA request from a XRCE Client that used DataFormat FORMAT_SAMPLE_SEQ.

The example illustrates the response to the request_id {0xAA, 0x02} from the XRCE DataReader with object_id {0x44, 0x06}. It uses the (already created) session with session_id 0xDD to send the data.

The data is sent using best-effort using the builtin stream identified by stream_id STREAMID_BUILTIN_BEST_EFFORTS.

This example also assumes the data being sent corresponds to a sequence of two objects ***foo1*** and ***foo1*** of type `FooType` defined in the IDL below In the example we assume foo1.count is set to 1 and foo2.count is set to 1.

```
@extensibility(FINAL)
struct FooType {
    long count;
};
```



The serialization of DATA_Payload_SampleSeq can be expanded as:

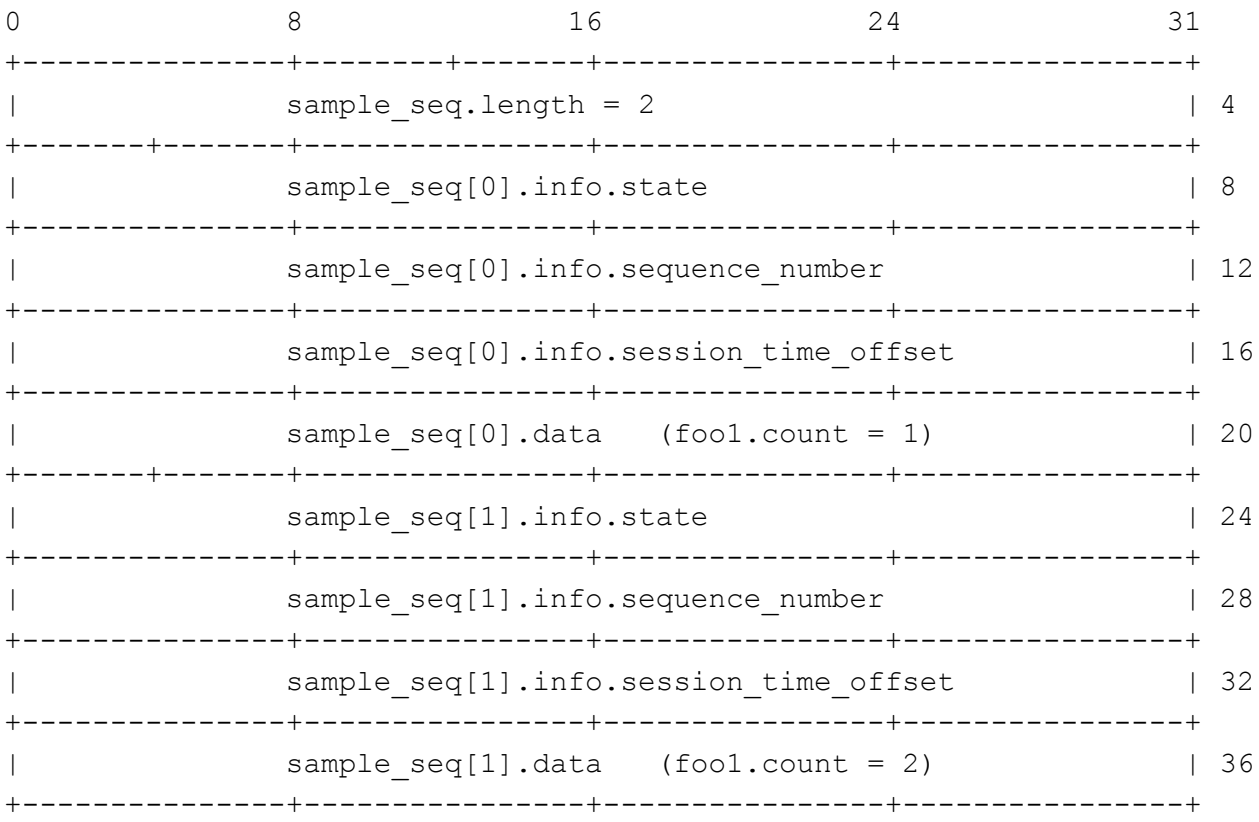


Table 35 Description of the DATA (sample sequence) example bytes

Bytes		Description
0-3	Message Header	
	Byte 0	sessionId = 0xDD
	Byte 1	streamId=0x01 Selects STREAMID_BUILTIN_BEST_EFFORTS, see 8.3.2.2
	Bytes 2-3	sequenceNr = 1 Represented in little endian (see 8.3.2.3) as {0x0A, 0x00}
4-7	Submessage Header	
	Byte 4	submessageId = DATA = 0x08
	Byte 5	flags = 0x00 (big endian)
	Bytes 6-7	submessageLength = 40 = 0x0028 Represented in little endian (see 8.3.4.3) as {0x28, 0x00}

8-47	DATA_Payload_SampleSeq (DataFormat was FORMAT_SAMPLE_SEQ)	
	Bytes 8-9	request_id = {0xAA , 0x01}
	Bytes 10-11	object_id = {0x44, 0x06}
	Bytes 12-15	sample_seq.length = 2
	Bytes 16-27	sample_seq[0].info
	Bytes 28-31	sample_seq[0].data
	Bytes 32-43	sample_seq[1].info
	Bytes 44-47	sample_seq[1].data

B.5.5. Receiving a sequence of packed samples

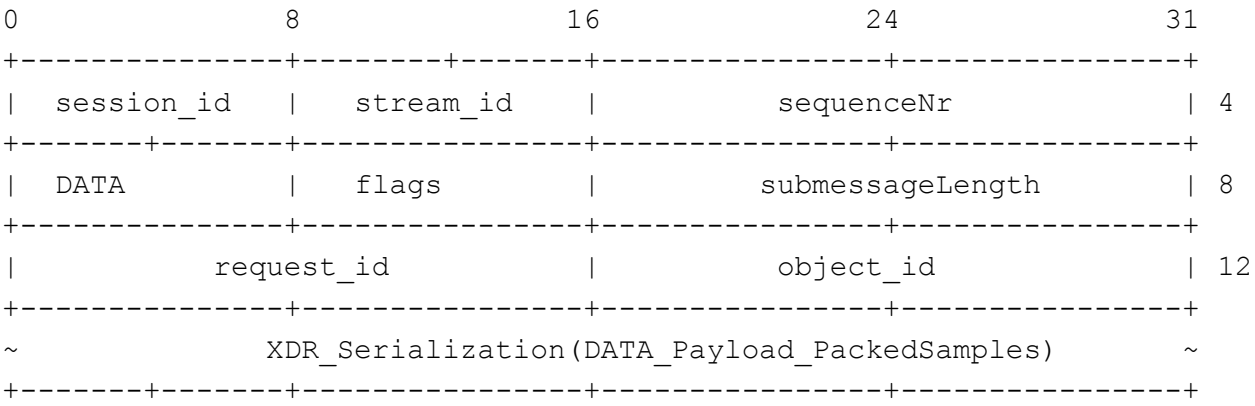
The following message could be used by a XRCE Agent to send a sequence of samples in response to a READ_DATA request from a XRCE Client that used DataFormat FORMAT_PACKED_SAMPLES.

The example illustrates the response to the request_id {0xAA, 0x03} from the XRCE DataReader with object_id {0x44, 0x06}. It uses the (already created) session with session_id 0xDD to send the data.

The data is sent using a reliable protocol using the builtin stream identified by stream_id STREAMID_BUILTIN_RELIABLE.

This example also assumes the data being sent corresponds to a sequence of two objects ***foo1*** and ***foo1*** of type `FooType` defined in the IDL below In the example we assume foo1.count is set to 1 and foo2.count is set to 1.

```
@extensibility(FINAL)
struct FooType {
    long count;
};
```



The serialization of DATA_Payload_SamplePackedSeq can be expanded as:

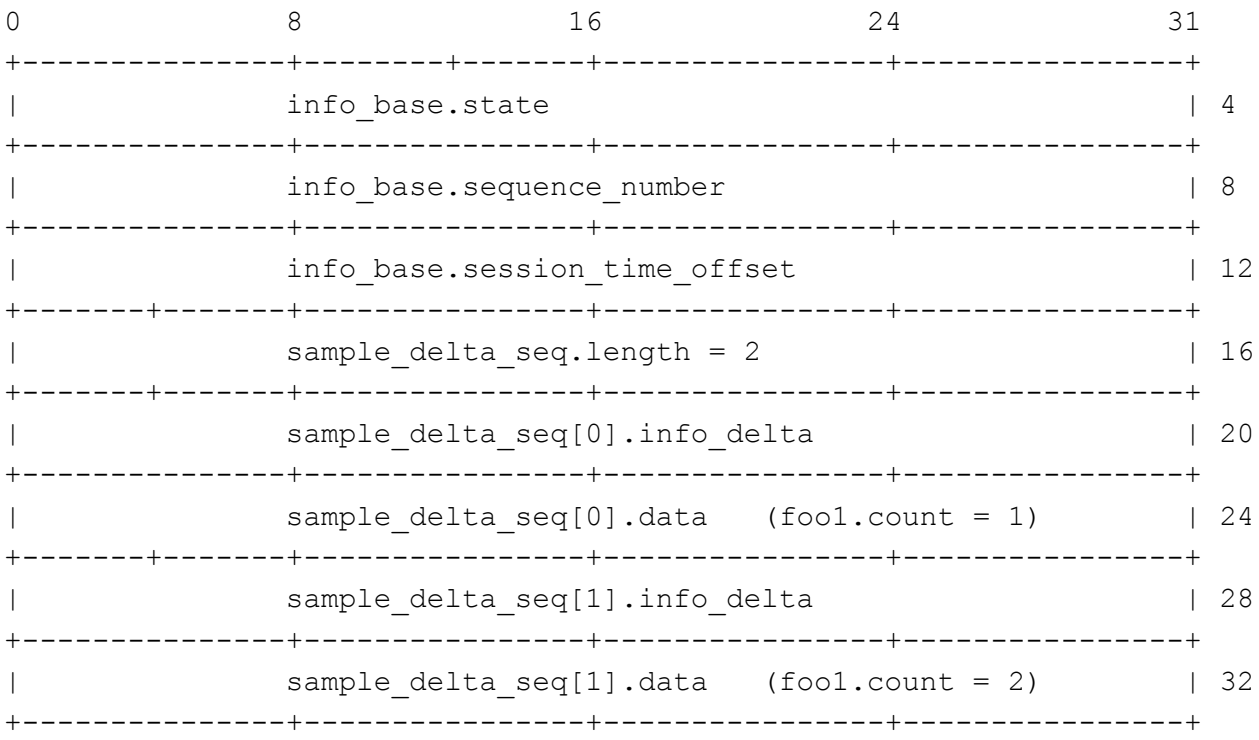


Table 36 Description of the DATA (packed samples) example bytes

Bytes		Description
0-3	Message Header	
	Byte 0	sessionId = 0xDD
	Byte 1	streamId=0x80 Selects STREAMID_BUILTIN_RELIABLE, see 8.3.2.2
	Bytes 2-3	sequenceNr = 1 Represented in little endian (see 8.3.2.3) as {0x0A, 0x00}
4-7	Submessage Header	
	Byte 4	submessageId = DATA = 0x08
	Byte 5	flags = 0x00 (big endian)
	Bytes 6-7	submessageLength = 36 = 0x0024 Represented in little endian (see 8.3.4.3) as {0x24, 0x00}
8-47	DATA_Payload_PackedSample (DataFormat FORMAT_PACKED_SAMPLES)	
	Byte 8-19	info_base

	Bytes 20-23	sample_delta_seq.length = 2
	Bytes 24-27	sample_delta_seq[0].info_delta
	Bytes 28-31	sample_delta_seq [0].data
	Bytes 32-35	sample_delta_seq [1].info_delta
	Bytes 36-39	sample_delta_seq [1].data