Generic Safe Computing Platform

OMG DDS Reference Implementation for Safe Computing Platform Messaging

Angel Martinez Bernal, Mark Carrier and Mark Hary, Real-Time Innovations (RTI)

Version 1.0, July 2022

Table of Contents

1 Introduction

1.1 About this Document

This document was produced in the context of the work on the "Specification of the Platform-Independent (PI) API between Application and Platform" [1] published in Open CCS Onboard Reference Architecture (OCORA) Release 2, which relates to the concept of a so-called Safe Computing Platform (SCP) [2] for future rail operation.

More precisely, this document elaborates on how the Object Management Group® (OMG®) Data Distribution Service™ (DDS®) could be used to realize the messaging among Applications as described in Section 6 in [1].

DDS is already well-proven in mission-critical systems in many industries, including smart transportation. Therefore, DDS appears as an interesting candidate for realizing the parts of the SCP PI API particularly relating to messaging.

The following sections explain how to represent the SCP components and interactions using DDS. This also includes the matching between SCP entities and DDS entities as well as the specification of the DDS Quality of Service (QoS) that these interactions should use.

This document provides guidance on implementing the SCP messaging API using DDS. Analysis and illustrations are included. No conclusion has been drawn as to whether SCP messaging could or should be based on any one solution or technology, including DDS.

1.2 Introduction to Data Distribution Service (DDS)

The Object Management Group® (OMG®) Data Distribution Service™ (DDS®) [3] family of standards [4] provides a platform-independent software framework used to architect and implement data-centric software solutions.

In data-centric communications, the focus is on the distribution of typed data between communicating applications. A data-centric system consists of data publishers and data subscribers. The communications are based on passing data of known types (data types) in named streams (Topics) from publishers to subscribers.

DDS also provides standardized mechanisms, known as Quality of Service (QoS) Policies, that allow applications to configure how communications occur, limit resources used by the middleware, detect system incompatibilities and set up error handling routines.

DDS allows every application to request 21 different QoS Policies such as deadlines, latency budgets, update frequencies, history, liveliness detection, reliability, ordering, filtering, and more. These QoS parameters allow system designers to construct a distributed application based on the requirements for, and availability of, each specific piece of data.

Examples include:

- *Durability* - Let late-joiners get data that was produced before they started.
- *Deadline* and *separation* - Specify min. and max. data update rates for each subscriber.
- *Liveliness* - Ensures that each dataflow is healthy.

- *Latency budget*, transport *priority*, & *reliability* - Decouple flow on a per-stream basis

Authentication, Access Control, Integrity, and Data Protection are also included as part of the DDS Specifications (DDS-SECURITY^{TM}) [5]. The security policies can be specified in a very granular way, providing ways to grant separate access by Domain Id, Topic name, and Data Tags. They also provide separate controls for read and write access. All mechanisms needed for access control, confidentiality, and integrity are provided. These include authentication handshakes, cryptographic key generation, secure key exchange, etc. data-level message-level encryption, addition of message authentication tags, etc.

Additionally, DDS defines an automatic discovery process in which different DDS entities find out about each other with no further configuration.

The knowledge of the data type that the applications exchange is fundamental to the data-centric publish-subscribe model and is required to implement several of the QoS Policies defined by DDS, including the DeadlineQosPolicy, HistoryQosPolicy, TimeFilterQosPolicy, ResourceLimitsQosPolicy. Typed information exchange is also required to notify of the Instance state (ALIVE, DISPOSED, or NO_WRITERS) of the various application-create data objects as well as implementing content-filtered topics. Data-type specification is also required to support safe-data exchange (semantic and serialized type compatibility between the publisher and subscriber types) as well as scalability and robustness in the presence of variations on the computer and network load as well as an intermittent network failure.

The Data-Types used by DDS are specified using the OMG Interface Definition Language^{TM} (IDL^{TM}) [6] (also an ISO/IEC standard [7]). Based on the IDL-specified type definitions the corresponding programming-level data-types are well defined by various OMG specifications (for example IDL-C++ [8] or IDL-CSHARP [9]), providing for application portability. Beyond portability, the data-serialization format is also specified by the DDS-XTypes^{TM} [10] standard, providing interoperability between components and applications implemented by different vendors. Additional benefits of these standards are that they provide dynamic ways to discover data-types, check type compatibility, and even support the evolution of data-types. All DDS vendors also provide a set of standard tools that are able to process the standard IDL format and generate all the necessary serialization and deserialization code.

For information on DDS and use-cases can be found at the DDS Foundation portal [11].

2 Definitions

The following definitions used in DDS are used throughout this document.

- **DomainParticipant**: DDS Entity used to join a DDS Domain. The DDS domain is identified by an integer domainId and represents a separate DataBus (or Data Space). The DomainParticipant is a factory for other DDS entities: publishers, subscribers and topics. It is responsible for discovering other DDS DomainParticipants and entities within the DDS Domain.
- **Topic**: A Topic represents an open collection of related data-objects that may be published or subscribed to. Each Topic may contain one or more data-objects, each identified by a different value of the Key field(s). The Topic is identified by a “topic name” that must be unique within its domain. Topics have an associated DataType which corresponds to the Type of the objects published or subscribed on that Topic.
- **Publisher**: DDS Entity that manages the activities of several DataWriters. It is used to create and group DataWriters.

- **Subscriber**: DDS Entity that manages the activities of several DataReaders. It is used to create and group DataReaders.
- **DataWriter**: DDS Entity used to write and publish data as well as receive notifications relevant to the data being published. It is bound at creation time to a Topic. A DataWriter may belong to only one publisher. This class is specialized for the concrete datatype that it is writing. Its behavior is controlled via QoS.
- **DataReader**: DDS Entity used to subscribe and read data as well as receive notifications relevant to the data being subscribed. It is bound at creation time to a Topic. A DataReader may belong to only one Subscriber. This class is specialized for the concrete datatype that it is reading. Its behavior is controlled via QoS.
- **Endpoints**: this encompasses DataWriters and DataReaders.
- **DataType**: this is the representation of the data that is transmitted between DataWriters and DataReaders. This data is structured and composed of one or more fields. This DataType may be represented using IDL. Fields may also contain different annotations which specify metadata of that specific field.
- **Key**: a key field is a member field of a DataType data type that is used to identify individual Data Objects- much like a primary key in a database. All key fields combined are called the Topic Key and their combined values provide a unique identifier of an individual Data-Object (called an instance) within a Topic.
- **Instance**: an instance is a unique Data-Object within a Topic, described by unique values of key fields. Instances represent individual real-world objects whose updates represent a single data stream. For example, you can represent individual trains within a single "TrainLocation" topic. What constitutes an instance will be different depending on what you are representing in your Topic. For example, a TemperatureSensor DataType has a field sensor_id that is marked as key. All sensors could publish their values in the same “Temperature” Topic. Since each sensor has a different value of sensor_id it will be considered a different Sensor that reports its temperature independently from the other Sensors.

3 Enhancements to the Proposed API

To enable some of the important QoS and data-centric capabilities in DDS, there may be some changes to the proposed APIs or at least some specific patterns on how these APIs are used. If this guidance is not followed, there may be a significant impact on the capabilities, performance or scalability of the resulting systems.

3.1 Usage of Typed Messaging APIs

As explained above, the DDS data-centric model is type-aware. This means that the created Topics must have an associated data-type that is registered with the DDS infrastructure.

The normal process is to define the Data Type using the standard IDL Language [7], or an equivalent XML syntax defined in the DDS-XML specification [12].

Note: from now on, this document assumes that IDL is used to represent the data types.

Once the IDL is created, specific code for that IDL shall be generated according to the OMG IDL mapping standards. This code includes:

- Language-specific definitions of the data type defined in the IDL

- Functionality to register the type within a DomainParticipant.
- Functionality to read/write data for this specific type.

One result of this process is the generation of type-specific APIs to read and write data of that type. The concrete API depends on the programming language being targeted. For example, if we assume types “Foo” and “Bar” are defined in the IDL and we are using C++, the code-generation process will define, the classes: DataWriter<Foo>, DataReader<Foo>, DataWriter<Bar>, and DataReader<Bar> that are used to publish and subscribe typed Foo and Bar, respectively. That is, these classes would provide DDS-standard API operations like:

```
DataWriter<Foo>::write(const Foo&);
DataReader<Foo>::read(Foo &, SampleInfo &);
DataWriter<Bar>::write(const Bar&);
DataReader<Bar>::read(Bar &, SampleInfo &);
```

Likewise, if the targeted language was C, then the corresponding functions would be:

```
FooDataWriter_write(FooDataWriter *, const Foo *);
FooDataReader_read(FooDataReader *, Foo *, SampleInfo *);
BarDataWriter_write(BarDataWriter *, const Bar *);
BarDataReader_read(BarDataReader *, Bar *, SampleInfo *);
```

Therefore, the natural mapping would be to use the generated functions to communicate different Functional Actors. This is the approach taken by other domain-specific specifications that are built on top of DDS such as Adaptive AUTOSAR and ROS 2.

These generated functions shall be used to implement the SCP API messaging functions: `fl_send and fl_receive.`

4 Platform Independent Model (PIM)

4.1 Introduction

The Platform Independent Model (PIM) describes the mapping and interactions between the SCP and DDS entities at an abstract (object model) level, independently of any programming Language APIs.

4.2 Functional Actors

The inputs and outputs of Functional Actors and their replicas are represented as DDS endpoints. A publisher of a Functional Actor for a specific topic or Flow is a DDS DataWriter, and a subscriber of a Functional Actor for a specific topic or Flow is a DDS DataReader.

4.3 Messaging

4.3.1 Introduction

This section specifies how to create SCP entities when using DDS. This will include uni-directional and bi-directional Flows. Also, this section defines the Quality of Services (QoSes) that shall be used in the DDS entities to accomplish the SCP requirements.

4.3.2 Functional Actor Authentication

In the case that authentication is needed between Functional Actors, the DDS DomainParticipants of all DataWriters and DataReaders involved in that communication must enable DDS-SECURITY with the following considerations:

The governance file shall disable unauthorized access for participants by setting the following options:

- allow_unauthenticated_participants to false.
- enable_join_access_control to true.
- discovery_protection_kind to *ENCRYPT_WITH_ORIGIN_AUTHENTICATION* or *SIGN_WITH_ORIGIN_AUTHENTICATION.*
- liveline*ss_protection_kind* to *ENCRYPT_WITH_ORIGIN_AUTHENTICATION* or *SIGN_WITH_ORIGIN_AUTHENTICATION*
- rtps_protection_kind to *ENCRYPT_WITH_ORIGIN_AUTHENTICATION* or *SIGN_WITH_ORIGIN_AUTHENTICATION*
- It shall contain *topic_access_rules* with one or more *topic_rule* entries:
    - Each *topic_rule* shall have at least the following elements:
    - The *topic_expression* may be a regular expression that is allowed in the *fnmatch()* function as specified in POSIX 1003.2-1992, Section B.6.
    - enable_discovery_protectionl to true.
    - enable_liveline*ss_protectionl* to true.
    - enable_read_access_control to true.
    - enable_write_access_control to true.
    - metadata_protection_kind to *ENCRYPT*, *SIGN, or NONE.*
    - data_protection_kind to *ENCRYPT* or *SIGN.*

4.3.3 Voting and Distribution of the Samples

The distribution of samples is done automatically by DDS as long as the entities are discovered correctly. DataWriters and DataReaders are correctly discovered if:

- Their corresponding DomainParticipant uses the same domainId.
- The topic name is the same
- They share the same PARTITION QoS name
- The datatype is compatible between them
- The QoSes are compatible between them

The voting process of the DDS samples is out of the scope of this specification.

4.3.4 Configuration of Flows

DataWriters and DataReaders must be created by the platform while creating/loading the configuration. These entities shall be created 'disabled' (*autoenable_created_entities* of the ENTITY_FACTORY QoS must be set to FALSE).

4.3.5 Uni-directional Flows

A uni-directional Flow in DDS is represented by a Topic. SCP publishers and subscribers can use a Flow, and they map to DDS DataWriters and DataReaders in a common PARTITION QoS. In this section, if a QoS is specified without a specific entity (DataReader or DataWriter), it must be applied to both entity kinds.

A Flow may be composed by:

- Exactly one SCP publisher: only one DataWriter is used with an unkeyed type.
- Multiple SCP publishers: data will be published using a keyed type. Each SCP publisher must use a different key value, hence they write different DDS instances, so they are clearly differentiated as different FAs.
    - There are different cases
        - All SCP publishers within the same process
            - A single DataWriter may be shared for all SCP publishers of the same Flow.
            - Alternatively, each SCP publisher may use a separate DDS DataWriter.
            - The DDS DataWriters may belong to the same DDS publisher or to different DDS publishers.
        - Different SCP publishers in different processes
            - Assuming that all Functional Actors are timely-synchronized.
            - Each SCP publisher will use its own DDS DataWriter
            - Each DataWriter must publish the data with different key values.

In these situations, the QoSes that DDS endpoints should follow are:

- Mandatory QoSes
    - RELIABILITY: depending on the configuration of the Flow:
        - At most once: BEST_EFFORT
        - At least once: RELIABLE
    - DESTINATION_ORDER:
        - Kind: BY_SOURCE_TIMESTAMP
    - LIVELINESS:
        - Kind: AUTOMATIC
        - Lease_duration: to the corresponding time when an entity is considered 'dead'.
    - LATENCY_BUDGET:
        - Duration
    - ENTITY_FACTORY

        - (Publishers and subscribers): autoenable_created_entities set to FALSE

- Optional QoSes
    - For periodic data, the following QoSes are defined:
        - HISTORY:
            - Kind: KEEP_LAST
            - Depth: 1
        - DURABILITY:
            - Kind: VOLATILE
    - In case that authentication must be performed within Functional Actors
        - Enable DDS Security as explained in the section 4.3.2.
    - In case it is needed to specify how long a data is valid:
        - (DataWriter QoS) LIFESPAN: duration
    - In case that knowing whether a sample is sent or received in a time window or when multiple SCP publishers in the same process share the same DDS DataWriter:
        - DEADLINE QoS: period

4.3.5.1 Key Characteristics of Uni-Directional Flows

This section describes how the key characteristics of Uni-directional Flows as introduced in [1] (in the following denoted in bold and italics) are accomplished with DDS.

Posted messages (on the same Flow and by the same publisher) are delivered to all subscribers in the exact same order as they have been published;

DataWriters send data with a corresponding sequence number. This allows DataReaders to identify when out-of-order data have been received. Depending on the Reliability kind QoS, the out-of-order data is discarded (BEST_EFFORT) or reordered (RELIABLE).

Therefore, all samples from the same DataWriter are delivered to all DataReaders in the exact same order, although they may be lost, depending on the Reliability kind QoS.

Missing messages are identified by the platform (e.g., through the usage of message sequence numbers or some other platform-specific mechanism). The subscribed FAs are notified by the Platform whenever there are missing messages.

DDS identifies if a sample is lost by setting the SAMPLE_LOST status. A DataReaderListener or a Waitset shall be configured with this status flag.

The DDS protocol includes the sequence number in every message and can detect sample lost, it might not be a 1-to-1 correspondence between sequence numbers and sample lost. In case a content filter is applied, data may be filtered out and not read by the application, but that doesn’t imply that the data sample has been lost. For this reason, it is not recommended to add sequence numbers to the sample itself, instead, applications should use the DDS sequence number and the notifications DDS provides.

Messages are time-stamped by the Platform, so that subscribers are able to determine how old messages are, and whether they should still process or discard them, etc.;

A DataWriter may set the LIFESPAN QoS policy to a specific duration. In this case, the messages whose lifespan expires will be considered lost and will be removed from the middleware caches and not delivered to the application.

DDS also provides timestamps in the SampleInfo available with each message received that may be used at the application level.

The RTE provides identification and authentication, so that from each message, subscriber(s) can determine and trust the identity of the publisher;
DDS-SECURITY must be enabled as specified in Section 4.3.2.

If desired by the subscriber(s) (e.g., via configuration or upon subscription to a Flow), subscriber(s) are notified when a publisher "dies" (meaning that it is either crashed or otherwise in a state where it cannot respond any more, or that it has not responded to past messages since a configured period of time).

An SCP publisher is considered “dead” when it has lost the connectivity (crash or not responding to messages). In this case, the LIVELINESS QoS shall be configured with the following parameters:

- Kind: AUTOMATIC
- Lease_duration: to the corresponding time when an entity is considered ‘dead’.

If the DataWriter’s *lease_duration* has been violated, that DataWriter is considered ‘dead’, then the LIVELINEES_CHANGED status change event will be triggered on the DataReader. A DataReaderListener or a WaitSet may be used to detect the change in the status.

If multiple SCP publishers are sharing the same DDS DataWriter, then the DataWriter should set the DEADLINE QoS policy and each SCP publisher should publish at least one message every DEADLINE period. If the SCP publisher dies, the DataReader will be notified via the DEADLINE_MISSED status change event that will be triggered in the DataReader. A DataReaderListener or a WaitSet may be used to detect the change in the status.

4.3.5.2 Static Properties of Uni-directional Flows

There are some static properties of Uni-directional Flows introduced in [1] that may imply changes in the QoS of the DataReaders and/or DataWriters. Those are specified in this section.

Message delivery options: Whether posted messages are received at most once or at least once;

These characteristics ARE met using the RELIABILITY QoS policy of the DataReaders and DataWriters:

- At most once: use DDS reliability kind BEST_EFFORT
- At least once: use DDS reliability kind RELIABLE

Desired maximum message delivery time (between publishing and receiving). It should be noted that this is not to be understood as a guarantee - subscribers should be able to use / compare time stamps to see if the maximum delivery time was fulfilled, and to decide whether messages should still be processed or discarded, etc.

Endpoints must set the QoS LATENCY_BUDGET and its duration to the desired time. Additionally, DataReaders may use timestamps available in the SampleInfo to calculate the latency of specific samples.

Likewise, if the Functional Actor requires to know when a sample is not sent (publisher) or received (subscriber) in a desired time window, they may set the DEADLINE QoS to a specific period.

4.3.5.3 Dynamic Properties of Uni-directional Flows

There is one dynamic property of Uni-directional Flows as introduced in [1] that Functional Actors should have access to.

Set of currently registered publishers and subscribers:

In order to identify which publishers or subscribers the Flow contain, the Functional Actor name shall be added to the DomainParticipant Property QoS:

- Name: "SCP_FA_name".
- Value: current value of the Functional Actor name.

Then, DataWriters and DataReaders should add the SCP publisher/subscriber identificator, as a Property QoS of the DataWriters and DataReaders:

- DataWriter
  - Name: "SCP_Publisher_id".
  - Value: current value that identifies the SCP publisher.
- DataReader
  - Name: "SCP_Subscriber_id".
  - Value: current value that identifies the SCP subscriber.

In case that a DataWriter is shared among different SCP publishers, the property "SCP_Publisher_id" shall contain a list of identifiers of all SCP publishers separated by the semicolon character ';'.

This information is retrieved by DataReaders and DataWriters by calling

- DataReaders:
  - *get_matched_publications()* and *get_matched_publication_data().*
- DataWriters:
  - *get_matched_subscriptions()* and *get_matched_subscription_data().*

4.3.5.4 Register and Unregister (Publisher) from a Uni-Directional Flow

In order to register as an SCP publisher in a uni-directional Flow, the corresponding DataWriter shall be enabled.

On the other side, if an SCP publisher is unregistered, the corresponding DataWriter should be deleted.

If for any reason, deletion of entities cannot be performed, each DataWriter shall belong to only one DDS Publisher. Then, the unregistration process consists of a change in the PARTITION QoS name to "UNREGISTERED_PUBLISHER". The PARTITION QoS is part of the DDS Publisher QoS, that is why every DataWriter shall belong to only one DDS Publisher.

If the SCP publisher is registered again, the PARTITION QoS name shall be modified to the original one (by default it is an empty string “”).

In the case where multiple SCP publishers running in the same process share the same DDS DataWriter, when an SCP publisher is unregistered it should “unregister” or “dispose” the DDS Topic instance with key corresponding to the SCP publisher.

4.3.5.5 Subscribe or unsubscribe from a Uni-Directional Flow

In order to subscribe to a uni-directional Flow, the corresponding DataReader shall be enabled.

On the other side, if the Functional Actor is unsubscribing from the Flow, the corresponding DataReader shall be deleted.

If for any reason, deletion of entities cannot be performed, each DataReader shall belong to only one DDS Subscriber. Then, the unsubscription process consists of a change in the PARTITION QoS name to “UNREGISTERED_SUBSCRIBER”. The PARTITION QoS is part of the DDS Subscriber QoS, that is why every DataReader shall belong to only one DDS Subscriber.

If the Functional Actor subscribes to the Flow again, the PARTITION QoS name shall be modified to the original one (by default it is an empty string “”).

Note: the PARTITION name on the publisher side and on the subscriber side is different, that is done on purpose to avoid matching between these endpoints.

4.3.6 Bi-directional Flows

A bi-directional Flow is implemented following the DDS-RPC^{TM} standard [13] request/reply style. That means that each node (requestor or replier) of the application has a DataWriter and a DataReader:

- Requestor node: It uses a DDS-RPC requester object consisting of:
    - A DataWriter sending the requests.
    - A DataReader receiving the replies.
- Replier node: It uses a DDS-RPC Replier object consisting of:
    - A DataReader receiving the request.
    - A DataWriter sending the replies.

Note that the DDS requester object is equivalent to the SCP requestor, and the DDS replier is equivalent to the SCP responder.

DDS-RPC requester and replier objects provide access to the underlying DataWriters and DataReaders. This may be needed in some situations, for example, when checking the status changes.

Additionally, depending on the architecture of the Flow, there may be some additional steps (for example how to apply *voting*):

1) **Neither requestor nor responder replicated, no voting applied**
    a) This is the common case specified by DDS-RPC. No changes.
2) **Only requestor replicated, voting applied on request path only**
    a) In this case, the request is voted. The voting process should happen before the responder reads the request message.
3) **Only responder replicated, voting applied on response path only**
    a) In this case, the reply is voted before the requestor application consumes it.

4) **Only responder replicated, no voting applied (note: mainly for debugging / logging purposes)**
   a) In this case, there are several responder applications (replicas) and the requestor application shall send the request to all responders and get the replies from all the responders.
5) **Both requestor and responder replicated, no voting applied ("bundling" scenario)**
   a) In this case, each interaction between the requestor and responder shall be done on a different PARTITION QoS. Each pair of requestor/responder replicas must use the same partition name.
   b) Other than that, the interaction between requestor and responder applications is the same as in bullet 1 above
6) **Both requestor and responder replicated, voting applied**
   a) In this case, the request and the response shall be voted before being consumed by the responder and requestor applications respectively.

4.3.6.1 Quality of Service

The underlying DataWriters and DataReaders must be configured with the following QoSes:

- Mandatory QoSes
    - RELIABILITY:
        - Kind: RELIABLE
    - DESTINATION_ORDER:
        - Kind: BY_SOURCE_TIMESTAMP
    - LIVELINESS:
        - Kind: AUTOMATIC
        - Lease_duration: to the corresponding time when an entity is considered 'dead'.
    - LATENCY_BUDGET:
        - Duration
    - DURABILITY:
        - Kind: VOLATILE
    - HISTORY:
        - Kind: KEEP_ALL
    - ENTITY_FACTORY
        - (Publishers and subscribers): autoenable_created_entities set to FALSE
- Optional QoSes
    - In case that authentication must be performed within Functional Actors
        - Enable DDS Security as explained in Section 4.3.2.
    - In case it is needed to specify how long a data is valid:
        - (DataWriter QoS) LIFESPAN: duration
    - In the "bundling" scenario:
        - (Publishers and subscribers) PARTITION QoS: name

4.3.6.2 Key characteristics of Bi-directional Flows

This section describes how the key characteristics of bi-directional Flows as introduced in [1] are accomplished with DDS.

Posted messages are received by the receiver in the exact same order as they have been sent. This applies to both messages sent by the requester, and the response messages sent by the responder;

Underlying DataWriters send data with a corresponding sequence number. This allows underlying DataReaders to identify when out-of-order data have been received. As this is using RELIABLE Reliability kind QoS, no out-of-order data is processed.

The Platform delivers messages (both requests and responses) exactly once (subject to a possible timeout);

The RELIABILITY QoS kind of the underlying DataReaders and DataWriters shall be configured to the RELIABLE.

The Platform, if so configured for the Flow, notifies the involved Functional Actors if the desired maximum message delivery time is exceeded. Also, this applies to both request and response messages;

The requester shall wait an amount of time to receive the reply. In case it is not received, it shall return an error code specifying the timeout. The DDS-RPC requester operations *reveive_replies()* and *wait_for_replies()* both take timeout parameters that may be used to detect that the message delay has been exceeded in the asynchronous or synchronous cases.

Additionaly, DDS identifies if a sample is lost by setting the SAMPLE_LOST status. A DataReaderListener or a Waitset shall be configured with this status flag on the requester side.

Messages are time-stamped by the Platform, so that the involved Functional Actors are able to determine how old messages are, and whether they should still process or discard them, etc.;

Underlying DataWriters may set the LIFESPAN QoS policy to a specific duration. In this case, the messages whose lifespan expires will be considered lost and will be removed from the middleware caches and not delivered to the application.

DDS also provides timestamps in the SampleInfo that may be used at the application level.

The Platform informs the requesting Functional Actor when the responding Functional Actor has joined the Flow (for the first time, or, e.g., after a crash)

The underlying endpoints of the requesting Functional Actor shall configure a Listener or a Waitset with the SUBSCRIPTION_MATCHED or PUBLICATION_MATCHED status for the underlying DataReader and DataWriter, respectively.

The Platform informs the requesting Functional Actor when the responding Functional Actor "dies" (meaning that it is either crashed or otherwise in a state where it cannot respond anymore, or that it has not responded to past messages since a configured period of time);

The Platform informs the responding Functional Actor when the requesting Functional Actor "dies" (meaning that it is either crashed or otherwise in a state where it cannot respond anymore, or that it has not responded to past messages since a configured period of time);

A requestor or responding Functional Actor is considered “dead” when it has lost the connectivity (crash or not responding to messages). In this case, the LIVELINESS QoS shall be configured with the following parameters:

- Kind: AUTOMATIC
- Lease_duration: to the corresponding time when an entity is considered ‘dead’.

If the DataWriter’s *lease_duration* has been violated, that DataWriter is considered ‘dead’, then the LIVELINEES_CHANGED status is enabled. A DataReaderListener or a Waitset may be used to detect the change in the state.

The Functional Actors involved in a Bi-directional Flow can trust the identity of the requesting / responding side.

DDS-SECURITY must be enabled as specified in the Section 4.3.2.

4.3.6.3 Static Properties of Bi-directional Flows

There is also one static property of Bi-directional Flows as introduced in [1] that may imply changes in the QoS of the DataReaders and/or DataWriters.

Desired maximum message delivery time (between sending and receiving). It should be noted that this is not to be understood as a guarantee - subscribers should be able to use / compare time stamps to see if the maximum delivery time was fulfilled, and to decide whether messages should still be processed or discarded, etc.

Underlying endpoints may set the QoS LATENCY_BUDGET and its duration to the desired time. Additionally, DataReaders may use timestamps to compare the latency for specific packages.

4.3.6.4 Joining / Disjoining Bi-directional Flows

In order to join on a bi-directional Flow, the corresponding underlying DataWriters and DataReaders shall be enabled.

A requestor on a bi-directional Flow must avoid sending any request if the state of the Flow is not either ‘CONNECTED’ or ‘DISCONNECTION_PENDING’.

To disjoin from a bi-directional Flow, the requestor and/or replier shall be deleted (including all underlying DDS entities).

If for any reason, deletion of entities cannot be performed, each DataWriter shall belong to only one DDS Publisher and each DataReader shall belong to only one DDS Subscriber. Then, the disjoining process consists of a change in the PARTITION QoS name to “DISJOINED_REQUESTER” or “DISJOINED_REPLIER”. The PARTITION QoS is part of the DDS Publisher/Subscriber QoS, that is why every endpoint shall belong to only one DDS Publisher/Subscriber.

If the SCP requester or responder joins again the Flow, the PARTITION QoS name shall be modified to the original one (by default it is an empty string “”).

References

[1] RCA/OCORA and industry partners, “Generic Safe Computing Platform - Specification of the PI API between Application and Platform”, OCORA-TWS03-030, Version 2.0, July 2022, see LINK
[2] RCA/OCORA, “An Approach for a Generic Safe Computing Platform for Railway Applications”, White Paper, Version 1.1, June 2021, see https://github.com/OCORA-Public/Publication/blob/master/05_OCORA R1/OCORA-TWS03-010_Computing-Platform-Whitepaper.pdf
[3] OMG, Data Distribution Service standard, see https://www.omg.org/spec/DDS/
[4] DDS Foundation, DDS Standards, see https://www.dds-foundation.org/omg-dds-standard/
[5] OMG, DDS Security standard, see https://www.omg.org/spec/DDS-SECURITY/
[6] OMG, Interface Definition Language standard, see https://www.omg.org/spec/IDL/
[7] ISO/IEC 19516:2020, see https://www.iso.org/standard/65379.html
[8] OMG, IDL to C++11 Language Mapping standard, see https://www.omg.org/spec/CPP11
[9] OMG, IDL4 to C# Language Mapping standard, see https://www.omg.org/spec/IDL4-CSHARP
[10] OMG, Extensible and Dynamic Topic Types for DDS standard, see https://www.omg.org/spec/DDS-XTypes/
[11] DDS Foundation Portal, see https://www.dds-foundation.org/
[12] OMG, DDS Consolidated XML Syntax, see https://www.omg.org/spec/DDS-XML/
[13] OMG, RPC Over DDS standard, see https://www.omg.org/spec/DDS-RPC/