Liveliness

11. marraskuuta 2013 8:33

8.4.13 Writer Liveliness Protocol
The DDS specification requires the presence of a liveliness mechanism. RTPS realizes this requirement with the Writer Liveliness Protocol. The Writer Liveliness Protocol defines the required information exchange between two Participants in order to assert the liveliness of Writers contained by the Participants.
All implementations must support the Wirter Liveliness Protocol in order to be interoperable.
8.4.13.1 General Approach
The Writer Liveliness Protocol uses pre-defined built-in Endpoints. The use of built-in Endpoints means that once a Participant knows of the presence of another Participant, it can assume the presence of the built-in Endpoints made available by the remote Participant and establish the association with the locally matching built-in Endpoints.
The protocol used to communicate between built-in Endpoints is the same as used for application-defined Endpoints.
8.4.13.2 Built-in Endpoints Required by the Writer Liveliness Protocol
The built-in Endpoints required by the Writer Liveliness Protocol are the BuiltinParticipantMessageWriter and BuiltinParticipantMessageReader. The names of these Endpoinst reflect the fact that they are general-purpose. These Endpoints are used for liveliness but can be used for other data in the future.
The RTPS Protocol reserves the following values of the EntityId_t for these built-in Endpoints:
ENTITYID_P2P_BUILTIN_PARTICIPANT_MESSAGE_WRITER
ENTITYID_P2P_BUILTIN_PARTICIPANT_MESSAGE_READER
The actual value for each of these EntityId_t instances is defined by each PSM.
8.4.13.3 BuiltinParticipantMessageWriter and BuiltinParticipantMessageReader QoS
For interoperability, both the BuiltinParticipantMessageWriter and BuiltinParticipantMessageReader use the following QoS values:
• reliability.kind = RELIABLE_RELIABILITY_QOS
• durability.kind = TRANSIENT_LOCAL_DURABILITY
• history.kind = KEEP_LAST_HISTORY_QOS
• history.depth = 1
8.4.13.4 Data Types Associated with Built-in Endpoints used by Writer Liveliness Protocol
Each RTPS Endpoint has a HistoryCache that stores changes to the data-objects associated with the Endpoint. This is also true for the RTPS built-in Endpoints. Therefore, each RTPS built-in Endpoint depends on some DataType that represents the logical contents of the data written into its HistoryCache.
DDS Interoperability Protocol, v2.1 119
Figure 8.26 defines the ParticipantMessageData datatype associated with the RTPS built-in Endpoint for the DCPSParticipantMessage Topic.
Figure 8.26 - ParticipantMessageData
8.4.13.5 Implementing Writer Liveliness Protocol Using the BuiltinParticipantMessageWriter and Builtin-ParticipantMessageReader
The liveliness of a subset of Writers belonging to a Participant is asserted by writing a sample to the BuiltinParticipantMessageWriter. If the Participant contains one or more Writers with a liveliness of AUTOMATIC_LIVELINESS_QOS, then one sample is written at a rate faster than the smallest lease duration among the Writers sharing this QoS. Similarly, a separate sample is written if the Participant contains ome or more Writers with a liveliness of MANUAL_BY_PARTICIPANT_LIVELINESS_QOS at a rate faster than the smallest lease duration among these Writers. The two instances are orthogonal in purpose so that if a Participant contains Writers of each of the two liveliness kinds described, two separate instances must be periodically written. The instances are distinguished using their DDS key, which is comprised of the participantGuidPrefix and kind fields. Each of the two types of liveliness QoS handled through this protocol will result in a unique kind field and therefore form two distinct instances in the HistoryCache.
In both liveliness cases the participantGuidPrefix field contains the GuidPrefix_t of the Participant that is writing the data (and therefore asserting the liveliness of its Writers).
The DDS liveliness kind MANUAL_BY_TOPIC_LIVELINESS_QOS is not implemented using the BuiltinParticipantMessageWriter and BuiltinParticipantMessageReader. It is discussed in Section 8.7.2.2.3.

8.7.2.2.3 LIVELINESS
Implementations must follow the approaches below:
• DDS_AUTOMATIC_LIVELINESS_QOS : liveliness is maintained through the BuiltinParticipantMessageWriter.

For a given Participant, in order to maintain the liveliness of its Writer Entities with LIVELINESS QoS set to AUTOMATIC, implementations must refresh the Participant's liveliness (i.e., send the ParticipantMessageData, see Section 8.4.13.5) at a rate faster than the smallest lease duration among the Writers.
• DDS_MANUAL_BY_PARTICIPANT_LIVELINESS_QOS : liveliness is maintained through the BuiltinParticipantMessageWriter. If the Participant has any MANUAL_BY_PARTICIPANT Writers, implementations must check periodically to see if write(), assert_liveliness(), dispose(), or unregister_instance() was called for any of them. The period for this check equals the smallest lease duration among the Writers. If any of the operations were called, implementations must refresh the Participant's liveliness (i.e., send the ParticipantMessageData, see Section 8.4.13.5).
• DDS_MANUAL_BY_TOPIC_LIVELINESS_QOS : liveliness is maintained by sending data or an explicit Heartbeat message with liveliness flag set.
The standard RTPS Messages that result from calling write(), dispose(), or unregister_instance() on a Writer Entity suffice to assert the liveliness of a Writer with LIVELINESS QoS set to MANUAL_BY_TOPIC. When assert_liveliness() is called, the Writer must send a Heartbeat Message with final flag and liveliness flag set.

9.6.2.1 Data Representation for the ParticipantMessageData Built-in Endpoints

The Behavior module within the PIM (Section 8.4) defines the DataType ParticipantMessageData. This type is the logical content of the BuiltinParticipantMessageWriter and BuiltinParticipantMessageReader built-in Endpoints.
The PSM maps the ParticipantMessageData tpe into the following IDL:

```
struct ParticipantMessageData {
  GuidPrefix_t participantGuidPrefix;
  octet [4] kind;
  sequence<octet> data;
};
```

The following values for the kind field are reserved by RTPS:

```
#define PARTICIPANT_MESSAGE_DATA_KIND_UNKNOWN {0x00, 0x00, 0x00, 0x00}
#define PARTICIPANT_MESSAGE_DATA_KIND_AUTOMATIC_LIVELINESS_UPDATE {0x00, 0x00, 0x00, 0x01}
#define PARTICIPANT_MESSAGE_DATA_KIND_MANUAL_LIVELINESS_UPDATE {0x00, 0x00, 0x00, 0x02}
```

RTPS also reserves for future use all values of the kind field where the most significant bit is not set. Therefore:
kind.value[0] & 0x80 == 0 // reserved by RTPS
kingd.value[0] & 0x80 == 1 // vendor specific kind
Implementations can decide the upper length of the data field but must be able to support at least 128 bytes.
Following the CDR encoding, the wire representation of the ParticipantMessageData structure is:

```
0...2...........8...............16.............24...............32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| unsigned long data.length                                     |
+---------------+---------------+---------------+---------------+
|                                                               |
~ octet[] data.value                                            ~
|                                                               |
+---------------+---------------+---------------+---------------+
```

Reliability

11. marraskuuta 2013 8:34

7.1.3.14 RELIABILITY

This policy indicates the level of reliability requested by a DataReader or offered by a DataWriter. These levels are ordered, BEST_EFFORT being lower than RELIABLE. A DataWriter offering a level is implicitly offering all levels below.

The setting of this policy has a dependency on the setting of the RESOURCE_LIMITS policy. In case the RELIABILITY kind is set to RELIABLE the write operation on the DataWriter may block if the modification would cause data to be lost or else cause one of the limits specified in the RESOURCE_LIMITS to be exceeded. Under these circumstances, the RELIABILITY max_blocking_time configures the maximum duration the write operation may block.

If the RELIABILITY kind is set to RELIABLE, data-samples originating from a single DataWriter cannot be made available to the DataReader if there are previous data-samples that have not been received yet due to a communication error. In other words, the service will repair the error and retransmit data-samples as needed in order to reconstruct a correct snapshot of the DataWriter history before it is accessible by the DataReader.

If the RELIABILITY kind is set to BEST_EFFORT, the service will not retransmit missing data-samples. However for data-samples originating from any one DataWriter the service will ensure they are stored in the DataReader history in the same order they originated in the DataWriter. In other words, the DataReader may miss some data-samples but it will never see the value of a data-object change from a newer value to an order value.

The value offered is considered compatible with the value requested if and only if the inequality "offered kind >= requested kind" evaluates to 'TRUE.' For the purposes of this inequality, the values of RELIABILITY kind are considered ordered such that BEST_EFFORT < RELIABLE.

a. For a RELIABLE DDS DataReader, changes in its RTPS Reader's HistoryCache are made visible to the user application only when all previous changes (i.e., changes with smaller sequence numbers) are also visible.
b. For a BEST_EFFORT DDS DataReader, changes in its RTPS Reader's HistoryCache are made visible to the user only if no future changes have already been made visible (i.e., if there are no changes in the RTPS Receiver's HistoryCache with a higher sequence number).

There may be scenarios where an alive Reader becomes unresponsive and will never acknowledge the Writer. Instead of blocking on the unresponsive Reader, the Writer should be allowed to deem the Reader as 'Inactive' and proceed in updating its queue. The state of a Reader is either Active or Inactive. Active Readers have sent ACKNACKs that have been recently received. The Writer should determine the inactivity of a Reader by using a mechanism based on the rate
DDS Interoperability Protocol, v2.1 123
and number of ACKNACKs received. Then samples that have been acknowledged by all Active Readers can be freed, and the Writer can reclaim those resources if necessary. Note that strict reliability is not guaranteed when a Reader becomes Inactive.

"DCPSSubscription," "DCPSPublication," and "DCPSTopic" Topics.
According to the DDS specification, the reliability QoS for these built-in Entities is set to 'reliable.'

ReliabilityKind_t:

```
struct {
    long value;
} ReliabilityKind_t;
```

Mapping of the reserved values:

```
#define BEST_EFFORT 1
#define RELIABLE 3
```

Resource limits

26. marraskuuta 2013 8:13

7.1.3.19 RESOURCE_LIMITS
This policy controls the resources that the Service can use in order to meet the requirements imposed by the application and other QoS settings.
If the DataWriter objects are communicating samples faster than they are ultimately taken by the DataReader objects, the middleware will eventually hit against some of the QoS-imposed resource limits. Note that this may occur when just a single DataReader cannot keep up with its corresponding DataWriter. The behavior in this case depends on the setting for the RELIABILITY QoS. If reliability is BEST_EFFORT, then the Service is allowed to drop samples. If the reliability is RELIABLE, the Service will block the DataWriter or discard the sample at the DataReader 28in order not to lose existing samples.
The constant LENGTH_UNLIMITED may be used to indicate the absence of a particular limit. For example setting max_samples_per_instance to LENGH_UNLIMITED will cause the middleware to not enforce this particular limit.
The setting of RESOURCE_LIMITS max_samples must be consistent with the max_samples_per_instance. For these two values to be consistent they must verify that “max_samples >= max_samples_per_instance.”
The setting of RESOURCE_LIMITS max_samples_per_instance must be consistent with the HISTORY depth. For these two QoS to be consistent, they must verify that “depth <= max_samples_per_instance.”
An attempt to set this policy to inconsistent values when an entity is created via a set_qos operation will cause the operation to fail.

28.So that the sample can be resent at a later time.

History

26. marraskuuta 2013 8:16

7.1.3.18 HISTORY
1. This policy controls the behavior of the Service when the value of an instance changes before it is finally
communicated to some of its existing DataReader entities.
2. If the kind is set to KEEP_LAST, then the Service will only attempt to keep the latest values of the instance and
discard the older ones. In this case, the value of depth regulates the maximum number of values (up to and
including the most current one) the Service will maintain and deliver. The default (and most common setting) for
depth is one, indicating that only the most recent value should be delivered.
3. If the kind is set to KEEP_ALL, then the Service will attempt to maintain and deliver all the values of the instance
to existing subscribers. The resources that the Service can use to keep this history are limited by the settings of the
RESOURCE_LIMITS QoS. If the limit is reached, then the behavior of the Service will depend on the
RELIABILITY QoS. If the reliability kind is BEST_EFFORT, then the old values will be discarded. If reliability
is RELIABLE, then the Service will block the DataWriter until it can deliver the necessary old values to all
subscribers.
The setting of HISTORY depth must be consistent with the RESOURCE_LIMITS max_samples_per_instance. For these
two QoS to be consistent, they must verify that depth <= max_samples_per_instance.