# DURABILITY QoS Parameter

return to the Quality of Service (QoS) Parameters

Note

> The decoupling between DataReader and DataWriter offered by the Publish/Subscribe paradigm allows an application to write data even if there are no current readers on the network. Moreover, a DataReader that joins the network after some data has been written could potentially be interested in accessing the most current values of the data as well as potentially some history.

The DURABILITY QoS policy controls whether the Service will actually make data available to late-joining readers. Note that although related, this does not strictly control what data the Service will maintain internally. That is, the Service may choose to maintain some data for its own purposes (e.g., flow control) and yet not make it available to late-joining readers if the DURABILITY QoS policy is set to VOLATILE.

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 DURABILITY kind are considered ordered such that `VOLATILE < TRANSIENT_LOCAL < TRANSIENT < PERSISTENT`.

For the purpose of implementing the DURABILITY QoS kind TRANSIENT or PERSISTENT, the service behaves “as if” for each `Topic` that has TRANSIENT or PERSISTENT DURABILITY kind there was a corresponding “built-in” `DataReader` and `DataWriter` configured to have the same DURABILITY kind. In other words, it is “as if” somewhere in the system (possibly on a remote node) there was a “built-in durability `DataReader`” that subscribed to that Topic and a “built-in durability `DataWriter`” that published that Topic as needed for the new subscribers that join the system.

For each `Topic`, the built-in fictitious “persistence service” `DataReader` and `DataWriter` has its QoS configured from the `Topic` QoS of the corresponding `Topic`. In other words, it is “as-if” the service first did find_topic to access the `Topic`, and then used the QoS from the Topic to configure the fictitious built-in entities.

A consequence of this model is that the transient or persistence serviced can be configured by means of setting the proper QoS on the Topic.

For a given Topic, the usual request/offered semantics apply to the matching between any DataWriter in the system that writes the Topic and the built-in transient/persistent DataReader for that Topic; similarly for the built-in transient/persistent DataWriter for a Topic and any DataReader for the Topic. As a consequence, a DataWriter that has an incompatible QoS with respect to what the Topic specified will not send its data to the transient/persistent service, and a DataReader that has an incompatible QoS with respect to the specified in the Topic will not get data from it.

Incompatibilities between local `DataReader`/`DataWriter` entities and the corresponding fictitious “built-in transient/persistent entities” cause the REQUESTED_INCOMPATIBLE_QOS/OFFERED_INCOMPATIBLE_QOS status to change and the

corresponding Listener invocations and/or signaling of `Condition` and `WaitSet` objects as they would with non-fictitious entities.

The setting of the service_cleanup_delay controls when the TRANSIENT or PERSISTENT service is able to remove all information regarding a data-instances. Information on a data-instances is maintained until the following conditions are met:

1. The instance has been explicitly disposed (instance_state = NOT_ALIVE_DISPOSED)
2. While in the NOT_ALIVE_DISPOSED state the system detects that there are no more “live” `DataWrite` entities writing the instance, that is, all existing writers either unregister the instance (call unregister) or lose their liveliness
3. And finally, a time interval longer that service_cleanup_delay has elapsed since the moment the service detected that the previous two conditions were met.

The utility of the service_cleanup_delay is apparent in the situation where an application disposes an instance and it crashes before it has a chance to complete additional tasks related to the disposition. Upon re-start the application may ask for initial data to regain its state and the delay introduced by the service_cleanup_delay will allow the restarted application to receive the information on the disposed instance and complete the interrupted tasks.

Source: DDS 1.4 Spec