TWINOAKS COMPUTING INC.
PRACTICAL MIDDLEWARE EXPERTISE

CoreDX DDS

Sample and Instance Management

Contents

This document describes the management of samples and instances in CoreDX DDS.

Sample and Instance Management in CoreDX DDS

Introduction

The concept of **samples** and **instances** is an important aspect of the DDS middleware technology and it impacts almost every feature within the DDS architecture. This document discusses the management of **samples** and **instances** within CoreDX DDS.

A **sample** is data of the appropriate DDS data type that has been published to a DDS Topic. A DDS data type may have a **key** which is composed of one or more data fields. The CoreDX DDS middleware uses the key to organize the published data. An **instance** is a collection (which could be empty) of samples with the same value in the **key** field(s). An **instance** is uniquely identified by an **instance handle**.

The Data Cache

Each DataReader and DataWriter contains its own Data Cache for storing samples and instances. A DataWriter Cache contains samples and instances that the DataWriter has published. A DataReader Cache contains samples and instances that the DataReader has received (subject to filtering and other QoS Policy settings). In general, these samples and instances are removed when they are no longer needed by the application or the CoreDX DDS middleware. The specific management of samples and instances in the Data Caches is described in the following sections.

Overview of Publisher Management

On the publishing side of CoreDX DDS communications, **samples** represent data that may be sent to DataReaders. Samples are created for every write(), unregister(), and dispose() call made by the application. Each **sample** is associated with a particular **instance**. In general, samples are added to the DataWriter Cache by write(), unregister_instance(), and dispose() calls made by the application. In general, instances are added to the DataWriter Cache (if they are not already there) by register_instance(), write(), unregister_instance(), and dispose() calls made by the application.

In general, samples and instances are stored in the DataWriter Cache until they are delivered to all appropriate DataReaders, at which point the samples and instances may be removed from the cache. The specific rules for maintaining and deleting samples from the DataWriter Cache are different from the rules for maintaining and deleting instances. For this reason, it is possible for all samples on an instance to be removed from the cache, while the instance remains (with no associated samples). In contrast, it is *not* possible to remove an instance from the cache while any samples associated with it remain.

Overview of Subscriber Management

On the subscribing side of CoreDX DDS communications, **samples** represent the data that has been received by the middleware and may be made available to the subscribing application (filters or other

QoS policy settings may preclude samples from reaching the application). Each received sample is associated with a particular **instance**. In general, samples and their associated instances are added to the DataReader Cache as they are received by the middleware.

In general, samples and instances are stored in the DataReader Data Cache until they are explicitly removed by the subscribing application, or the CoreDX DDS middleware removes them based on various QoS policy settings. Similar to the DataWriter Data Cache management, the specific rules for maintaining and deleting samples from the DataReader Data Cache are different from the rules for maintaining and deleting instances. For this reason, it is possible for all samples on an instance to be removed from the cache, while the instance remains (with no associated samples). In contrast, it is *not* possible to remove an instance from the cache while any samples associated with it remain.

Instance Lifecycles

Instances are used to manage the DDS data lifecycle. Instances are created, either explicitly by the application, or automatically when a sample on the instance is written. Instances are updated when additional samples on the instance are written. Instances are (logically) deleted, either explicitly by the application, or automatically when the DataWriter (or DataWriter's application) exits. These data lifecycle operations play an important role in the management of both the DataWriter and matched DataReaders Caches. This section provides an overview of the instance lifecycle within CoreDX DDS, including general effects on the Data Caches. The specific events within the Data Caches depend on several DataWriter and DataReader QoS policy settings, and are documented in later sections of this document.

Registering Instances

Every data sample that is published belongs to an instance. If the data type does not have a key specified, then every sample that is published belongs to the *same* instance. If the data type has a key specified, then each sample is grouped and assigned an instance based on the value of that key field(s). Instances must be **registered** with the DataWriter before any samples associated with that instance can be written (or deleted).

When a publishing application registers an instance with a DataWriter, the *instance* is added to that DataWriter Cache. However, a *sample* will **not** be added to the DataWriter Cache. An instance must be registered with the DataWriter before write(), unregister_instance(), or dispose() can be called for that instance. Publishing applications can use the DataWriter::register_instance() operation to *explicitly* register instances. As a convenience, CoreDX DDS will *automatically* register an instance when the application calls one of the write(), unregister_instance(), or dispose() operations without first registering the instance. When this happens, both a sample and an instance are added to the DataWriter Cache: an instance for the implicit register operation, and a sample for the written sample, unregister, or dispose.

Register instance operations are applicable only to DataWriters, and are not communicated to DataReaders. A DataReader does not have an interface to explicitly register instances. Instead, instances are created in the DataReader cache automatically based on the reception of samples. When a DataReader receives a sample, a sample is added to its Data Cache (if possible based on QoS policies). If the sample's associated instance is new to this DataReader, an instance will also be added to its DataReader Cache.

Unregistering Instances

The publishing application can **unregister** a previously registered instance by calling DataWriter::unregister_instance(). This indicates the application will no longer write any samples on this instance with this DataWriter. This is not the same as *disposing* an instance, which is described below.

When a publishing application unregisters an instance with a DataWriter, CoreDX DDS may remove the instance (and all related samples) from that DataWriter Cache. The actual removal from its Data Cache may be delayed, depending on QoS policy settings. For example, a DataWriter configured with Reliability.kind = RELIABLE will not remove an unregistered instance (and samples) from its Data Cache until all matched, Reliable DataReaders have successfully acknowledged all samples for the instance.

After an instance has been unregistered from that DataWriter, the **instance handle** associated with that instance is invalid. This is because the CoreDX DDS middleware may have removed all records of that instance. After an unregister operation, the instance handle may be reused for a different instance. If necessary, the application may re-register the instance (obtaining a new handle for it) and then continue to publish samples or a dispose on the instance.

Unregister operations are communicated to matched DataReaders and indicate that the DataWriter is no longer actively writing on this instance. The DataReader will remove this DataWriter from the instance's list of active DataWriters. When this list of alive, actively writing DataWriters becomes empty, the state of the instance in the DataReader Cache will change to NOT_ALIVE_NO_WRITERS.

When a DataReader receives an unregister command, it creates a sample and (if this is the first sample for this instance) an instance in its Data Cache. In general, the unregister command is treated like a data sample: it stays in the DataReader Cache until it is *taken* by the subscribing application. However, there are QoS policy settings that may trigger automatic processing by the CoreDX DDS middleware when the sample expires or the instance state changes to NOT_ALIVE_NO_WRITERS. If enabled, this automatic processing may automatically purge the instance and related samples from the DataReader Cache.

When a DataWriter is deleted by the publishing application, CoreDX DDS will automatically send an unregister command to matched DataReaders for every instance in that DataWriter Cache. If the publishing application exits without deleting its DataWriter Entities, the DataReader will not receive the unregister commands. In this case, the DataReader will eventually determine that the DataWriter is not alive, and then remove the DataWriter from the list of alive and actively writing DataWriters on each instance in its DataReader Cache.

Disposing Instances

The publishing application can **dispose** a previously registered instance by calling DataWriter::dispose(). The standards define a dispose operation as an indication that the instance no longer exists. However, this is an application level concept, and has very little meaning to the CoreDX DDS middleware. CoreDX

DDS treats a dispose very much like an application written data sample. The dispose command is added to the DataWriter Cache as a sample. If the instance for this sample is not already registered, CoreDX DDS will automatically register it before adding the dispose sample to its Data Cache.

Unlike the unregister command, the dispose command does not cause the DataWriter to perform any special management of the instance in its Data Cache.

Dispose operations are communicated to matched DataReaders. The DataReader will change the state of the associated instance in its Data Cache to NOT_ALIVE_DISPOSED. Note that the DataWriter will still be considered an alive, actively writing DataWriter on this instance.

When a DataReader receives a dispose command it creates a sample and (if this is the first sample for this instance) an instance in its Data Cache. In general, the dispose command is treated like a data sample: it stays in the DataReader Cache until it is *taken* by the subscribing application. However, there are QoS policy settings that may trigger automatic processing by the CoreDX DDS middleware where the sample expires or the instance state changes to NOT_ALIVE_DISPOSED. If enabled, this automatic processing may automatically purge the instance and related samples from the DataReader Cache.

Instance Handles

An **instance handle** is a value that can be used to uniquely identify a *registered* instance. An instance handle is generated when an instance is registered (returned from a DataWriter::register_instance() call) and will be used to identify the instance until that instance is unregistered. ***The instance handle is valid only while the instance is registered***. Once an instance is unregistered, the instance handle can no longer be used to identify that instance. This is a critical detail of the CoreDX DDS middleware. Instance handles may be reused, and, after an unregister operation the old instance handle may identify a *different* instance. If the unregistered instance is re-registered, a different handle may be assigned for the next 'life' of that instance.

Liveliness (of DataWriters)

CoreDX DDS maintains a concept of Liveliness that is applied to DataWriters. A DataWriter may be considered alive for a number of reasons:

a. The DataWriter is actively writing data (and has not missed any deadlines as configured via the Deadline QoS policy)
b. The DataWriter is related to other DataWriters (by having the same publisher) that are alive
c. The DataWriter belongs to an DomainParticipant that is alive, or
d. The publishing application explicitly calls DataWriter::assert_liveliness() periodically

Depending on the configuration of the DataWriter's Liveliness QoS policy, failure to meet one or more of the above conditions may result in the DataWriter becoming *not* alive. The liveliness of a DataWriter can have an effect on Data Caches of matched DataReaders.

DataReader Caches contain a list of matched, alive, and actively writing DataWriters for each instance in the Cache. If an instance has at least one of these matched, alive, and actively writing DataWriters, the instance state is ALIVE. If an instance does not have any of these matched, alive, and actively writing

DataWriters, the instance state is NOT_ALIVE_NO_WRITERS. This status, along with the configuration of other QoS policies described below effect the management of the instance and associated samples in the DataReader Cache.

Effect of Qos Policy Settings

Data caches are sized and managed according to the configuration of several QoS policies. The following table provides an overview of the QoS Policies that affect Data Cache Management, along with where they are set and which Data Caches (DataReader, DataReader, or both) they affect.

QoS Policy	Configuring this policy on the:	Will effect Cache Management on the:
RELIABILITY	DataWriter and DataReader DataReader	DataWriter DataReader
DURABILITY	DataWriter	DataWriter and DataReader
HISTORY	DataWriter DataReader	DataWriter DataReader
RESOURCE_LIMITS	DataWriter DataReader	DataWriter DataReader
READER_DATA_LIFECYCLE	DataReader	DataReader
WRITER_DATA_LIFECYCLE	DataWriter	DataReader
OWNERSHIP	DataWriter	DataReader
LIFESPAN	DataWriter	DataWriter and DataReader
Filters (TIME_BASED_FILTER, content filters)	DataReader	DataReader

Table 1

Reliability

The Reliability QoS policy configures the level of reliability CoreDX DDS will guarantee for communications between a DataReader and DataWriter. With a *Best Effort* Reliability configuration, CoreDX DDS will make an effort to deliver all published data, but there is no guarantee all data will be received by all matched DataReaders. With a *Reliable* configuration, CoreDX DDS will guarantee all published data will be received by all DataWriters.

Reliability and the DataWriter Cache

Both *Reliable* and *Best Effort* DataWriters add samples and instances to their Data Caches in the same way, under the same circumstances. A sample is added when it is written (DataWriter::write() is called), when an instance is unregistered (DataWriter::unregister_instance() is called), and when an instance is disposed (DataWriter::dispose() is called). Instances are added to DataWriter Caches (both *Reliable* and *Best Effort* DataWriters) when they are registered.

A *Best Effort* DataWriter will remove samples from its Data Cache as soon as they are written on the wire. Instances are kept in the DataWriter cache until the application explicitly unregisters them. At that time, the instance and its associated samples (if any) are removed from that DataWriter Cache.

A *Reliable* DataWriter will keep samples in its Data Cache until they have been acknowledged by all matched *Reliable* DataReaders. [*Best Effort* DataReaders are not required to acknowledge samples.] Once a sample has been acknowledged by all matched Reliable DataReaders, it will be removed from the cache. Instances are kept in the DataWriter Cache until the application explicitly unregisters them. The instance will be removed once all associated samples have been removed (samples are removed when they have been received and acknowledged by matched, *Reliable* DataReaders).

Reliability and the DataReader Cache

The Reliability QoS policy has a more subtle effect on DataReader Caches. In general, DataReaders add samples and instances to their Data Cache as they are received. However, *Reliable* DataReaders and *Best Effort* DataReaders matched with the same DataWriters may have different samples and instances in their Data Caches.

With a *Best Effort* DataReader, samples may be missed. Because the DataReader is Best Effort, these samples will not be recovered. Missed data samples are data samples that are not added to the DataReader Cache. Missed unregister commands are samples that are not added to the DataReader Cache, and furthermore, they will cause the state of the instance to remain ALIVE, when it is possible that the instance should be NOT_ALIVE_NO_WRITERS. Similarly, missed dispose commands are samples that are not added to the DataReader Cache, and furthermore, they will cause the state of the instance to remain ALIVE, when that instance should be NOT_ALIVE_DISPOSED.

Durability

The Durability QoS Policy configures how long data will be saved by the DataWriter, in order to make it available to late joining DataReaders. [A 'late joining' DataReader is a DataReader that is enabled after a DataWriter has published some data samples.] The publish-subscribe paradigm offered by CoreDX DDS allows applications to write data even when there are no current readers on the network. Further, a DataReader has the option to receive historical data (data published before this DataReader was enabled) in addition to currently published data. The Durability policy supports this configuration.

Volatile DataWriters will not save previously published data for late joining readers. *Transient Local* DataWriters that are also *Reliable* will save previously published data for the life of the DataWriter and make this data available to late joining DataReaders. The additional Durability kinds (Transient and Persistent) are not currently supported by CoreDX DDS.

Durability and the DataWriter Cache

The *Volatile* Durability kind does not affect the DataWriter Cache management – the behavior is determined by only by the other QoS policies documented here.

A *Transient Local* DataWriter may delete instances from its Data Cache only when the publishing application unregisters them by calling DataWriter::unregister_instance(). When this happens, a *Reliable Transient Local* DataWriter will wait for all currently matched DataReaders to acknowledge all samples associated with this instance before removing the instance from the Data Cache. Similarly, a *Best Effort Transient Local* DataWriter will wait until all current samples have been written onto the wire before removing the instance.

For *Transient Local* DataWriters, samples may be deleted from a DataWriter Cache only when the associated instance is deleted from the Data Cache, a sample expires due to Lifespan QoS policy settings, or the History QoS policy kind is set to KEEP_LAST.

Durability and the DataReader Cache

The Durability QoS policy has a more subtle effect on DataReader Caches. The Durability QoS policy does not affect when samples and instances are removed from the DataReader Cache, but it can affect which samples are added to the DataReader Cache. In general, DataReaders add samples and instances to their Data Cache as they are received. However, *Volatile* DataReaders and *Transient Local* DataReaders matched with the same DataWriters may have different samples and instances in their Data Caches.

A DataWriter may register, write samples on, dispose, and unregister an instance before a DataReader is created and matched with that DataWriter. If this DataReader is *Volatile*, these historical samples will not be sent, and will not be added to the DataReader Cache. If this DataReader is *Transient Local*, these historical samples will be sent and added to the DataReader Cache, resulting in samples and instances in the *Transient Local* DataReader Cache that are not in the *Volatile* DataReader Cache.

History

The History QoS policy controls the number of data samples CoreDX DDS will store and manage for each instance in the Data Cache. The History QoS policy controls the amount of buffering provided by both the DataWriter and DataReader. On a DataWriter, the History QoS policy determines the amount of data history that is preserved to be retransmitted to *Reliable* DataReaders or provided to late joining *Transient Local* DataReaders. On a DataReader, this policy will determine the number of samples available to return on a read() or take() operation.

In general, with a History kind of *KEEP_ALL*, CoreDX DDS must keep all samples in the Data Caches subject to other QoS policy configurations (for example, resource limits and reliability). In general, with a History kind of *KEEP_LAST*, CoreDX DDS may remove older samples for an instance in order to make room for newer samples on the same instance. With *KEEP_LAST*, the number of samples maintained in the Data Cache for each instance is specified by the History depth setting.

History and the DataWriter Cache

The DataWriter Cache contains samples and instances that have been written by the publishing application. DataWriters that are *Reliable* or *Transient Local* and *Reliable* may keep samples in the cache after they have been initially published. [A DataWriter that is *Best Effort,* or *Reliable* with no matched *Reliable* DataReaders, may remove samples from its Data Cache after they have been initially published on the network.] The History QoS policy settings on the DataWriter help determine how many of these samples are kept in the DataWriter Cache.

A *Reliable* and *Volatile* DataWriter with a History kind of *KEEP_ALL* will keep samples (subject to resource limits) in its Data Cache until all currently matched *Reliable* DataReaders have received and acknowledged the sample. At that time, the sample can be removed from the DataWriter Cache.

A *Reliable* and *Transient Local* DataWriter with a History kind of *KEEP_ALL* will keep samples (subject to resource limits) in its Data Cache until the publishing application unregisters the associated instance.

If either of these *Reliable KEEP_ALL* DataWriters runs out of room in its Data Cache (this could happen in combination with Resource Limits QoS policy settings if a *Reliable* DataReader is not acknowledging samples in a timely manner or with any *Transient Local* DataWriter if the application does not unregister instances), any operation that creates a sample (write(), unregister_instance(), dispose()) will either return an error or block the application. The amount of time a DataWriter will wait for room in its Data Cache is controlled by the *reliability.max_blocking_time* setting.

A DataWriter with a History *kind* of *KEEP_LAST* will keep no more than History *depth* samples for each instance in its Data Cache. If a DataWriter runs out of room in its Data Cache (under similar conditions as above), it will remove an old sample to make room for the new sample. The removed sample is no longer available for transmission to DataReaders.

History and the DataReader Cache

The DataReader Cache contains samples and instances that have been received by the middleware and may be made available to the subscribing application. The History QoS policy settings on the DataReader help determine how many of these samples are kept in the DataReader Cache.

In general, received samples are not removed from the DataReader Cache until the subscribing application calls DataReader::take(), the samples expire with the Lifespan QoS policy, or the associated instance is removed per Reader Data Lifecycle QoS policy settings. If the subscribing application does not use the take() operation, received samples can accumulate in the DataReader Cache.

A *Reliable* DataReader with a History kind of *KEEP_ALL* will not overwrite samples in its Data Cache. If this DataReader runs out of room in its Data Cache (this could happen in combination with Resource Limits QoS policy settings) any received samples that do not fit in the DataReader Cache are dropped and are not acknowledged to the DataWriter. [Refer to the discussion on Reliability for resulting behavior at the DataWriter.]

A DataReader with a History kind of KEEP_LAST will keep no more than History *depth* samples for each instance in its Data Cache. If a *Keep Last* DataReader runs out of room in its Data Cache (under similar conditions as above), it will remove an old sample to make room for the new sample. The removed sample is no longer available to the subscribing application.

Resource Limits

The Resource Limits QoS policy sets an upper bound on the number of samples and instances that can be stored in the DataReader or DataWriter Cache. The specific Resource Limits that can be configured are: *max samples* (the total number of samples in the Data Cache), *max instances* (the total number of instances in the Data Cache), and *max samples per instance* (the total number of samples associated with each instance in the Data Cache).

The Resource Limits and History QoS policies can be used in concert to configure the management of **samples** in the Data Caches (DataWriter and DataReader Caches). With a Reliability kind of *Reliable* and a History kind of *KEEP_ALL*, and *max samples* or *max samples per instance* Resource Limits configured (that is, not INFINITE), the CoreDX DDS middleware will not remove an old sample to make room for a new sample. With a History kind of KEEP_LAST and either *max samples* or *max samples per instance* Resource Limits configured, the middleware is allowed to remove older samples to make room for newer samples if necessary.

The Resource Limits QoS policy can also be used to configure the management of **instances** in the Data Caches (DataWriter and DataReader Caches). However, careful consideration is recommended before configuring the *max instances* Resource Limits, since there are configurations that can effectively deadlock a DataWriter from publishing any more samples, or a DataReader from processing any more samples from that DataWriter. For example, consider a *Reliable*, *Keep All* DataReader with *max instances* set to 10, matched with a DataWriter that has published samples on 11 instances. When the DataWriter publishes the first sample on the 11th instance, the DataReader will be unable to accept it, due to the *max instances* resource limits. The *Reliable* DataReader will not acknowledge this new sample (or any subsequent samples from the DataWriter), effectively blocking that DataWriter.

This is a problem that arises only with the QoS policy combination of *Reliable* Reliability, *Keep All* History, and *max instances!=INFINITE* Resource Limits. To eliminate this possible deadlock scenario with *max_instances*, it is recommended to always set *max instances* such that the DataWriter *max_instances* is <= the DataReader *max_instances*.

Resource Limits and the DataWriter Cache

Data Writers that are *Reliable* and *Keep All* have the potential to block or return an error when the publishing application attempts to add a sample or instance to a "full" DataWriter cache. An operation that creates an instance in the DataWriter Cache (an operation that results in registering a new instance) will fail or block when the *max_instances* Resource Limit is reached. An operation that attempts to add a sample to the DataWriter Cache (a write(), unregister_instance(), or dispose() operation) will fail or block the publishing application when the History kind is *KEEP_ALL* and the *max samples* or *max samples per instance* Resource Limit is reached. The amount of time these operations will block is controlled by the Reliability *max_blocking_time* setting.

DataWriters that are *Best Effort* or *Keep Last* will block only when the publishing application performs an action on the DataWriter that will create a new instance and the *max instances* Resource Limit has been reached. Both these types of DataWriters are able to remove an old sample to make room for a new one, so operations that just create a new sample will not block or return an error.

Resource Limits and the DataReader Cache

A DataReader that is *Reliable* and *Keep All* has the potential to *REJECT* received samples when its Data Cache is full. When *max samples* or *max samples per instance* limits are reached, any received sample will be rejected, until samples are removed from the DataReader Cache. When *max instances* limits are reached, any received sample *on that instance* will be rejected. A sample that is *rejected* is not acknowledged. This can impact the DataWriter that published the sample, if the DataWriter is configured as *Reliable* and *KEEP_ALL*. This DataWriter will not delete any samples (or related instances) from its Data Cache until all currently matched DataReader have acknowledged them.

DataReaders that are *Best Effort* or *Keep Last* are able to drop samples without impact to the publishing DataWriter. These dropped samples are acknowledged so the DataWriter will consider them to be delivered.

Reader Data Lifecycle

The Reader Data Lifecycle QoS policy directly controls the management of samples (and indirectly controls the management of instances) in the DataReader Cache. This QoS policy has no effect on the DataWriter Cache.

The Reader Data Lifecycle policy allows samples to be automatically removed from the DataReader Cache. When the *autopurge nowriter samples delay* is configured to non-INFINITE samples will be removed from the DataReader Cache when the associated instance state becomes NOT_ALIVE_NO_WRITERS (after the configured delay period). When the *autopurge disposed samples delay* is configured to non-INFINATE, samples will be removed from the DataReader Cache when the associated instance state becomes NOT_ALIVE_DISPOSED.

Instances will be removed from the DataReader Cache when there are no associated samples in the DataReader Cache and the instance state is NOT_ALIVE_NO_WRITERS. Because the Reader Data Lifecycle policy may remove all samples from an instance with a state of NOT_ALIVE_NO_WRITERS, it may also cause the instance to also be removed.

Writer Data Lifecycle

The Writer Data Lifecycle QoS policy controls the behavior of the DataWriter with regard to the lifecycle of instances it is maintaining. This QoS policy has one option: *auto-dispose unregistered instances* that can be set to TRUE or FALSE. When set to TRUE, CoreDX DDS will automatically dispose instances when they are unregistered by the publishing application. Since the dispose operation must have a registered instance to work on, it is applied first, and then the unregister operation is applied to the instance.

When a DataWriter is deleted by the publishing application, an unregister operation is automatically applied to all the instances in that DataWriter Cache. When the *auto-dispose unregistered instances* option is set to TRUE, a dispose operation is also automatically applied to all instances in that DataWriter Cache. [Again, the dispose operation is applied before the unregister operation.]

Writer Data Lifecycle and the DataWriter Cache

When the publishing application unregisters an instance on a DataWriter with its Writer Data Lifecycle QoS policy configured to automatically dispose unregistered instances, the DataWriter will create a sample (and add it to its Data Cache) for the user-invoked unregister instance operation. If *autodispose_unregistered_instances* is TRUE, CoreDX DDS will combine a dispose operation with the unregister operation. [An additional sample is ***not*** created (and not added to its Data Cache) for the automatic dispose operation.]

Writer Data Lifecycle and the DataReader Cache

If *autodispose_unregistered_instances* is TRUE at the DataWriter, CoreDX DDS will send a combined *dispose* and *unregister* sample to matched DataReaders. The combined *dispose* and *unregister* sample is added to the instance.

Ownership

The Ownership QoS policy controls whether CoreDX DDS will allow multiple DataWriters to update the same instance at the same time. The possible values for Ownership are: *Shared* and *Exclusive*. When set to *Shared*, CoreDX DDS does not enforce unique ownership for each instance, and multiple DataWriters can update the same instance at the same time. When set to Exclusive, each instance can be modified by only one DataWriter. In this case, each instance has only one DataWriter that is considered the owner, and while that DataWriter is "alive", it is the only writer allowed to update the instance.

This Ownership management is applied at the DataReaders, and affects the DataReader Cache. It has no effect on the DataWriter Cache.

An *Exclusive Ownership* DataReader will still maintain a list of alive and actively writing DataWriters for each instance, but only one of those DataWriters can be the current owner. Each instance has its own current DataWriter owner. Samples received from any DataWriter other than the owner for that instance will be discarded, and are not added to the DataReader Cache.

Lifespan

The Lifespan QoS policy allows CoreDX DDS to expire old data samples. The Lifespan QoS policy has a *duration* which is the amount of time that may pass after a sample has been published before the sample is considered expired.

Lifespan and the DataWriter Cache

Samples that are expired by the Lifespan QoS policy configuration are removed from the DataWriter cache. This is applicable for *Reliable* DataWriters: samples that are in the DataWriter Cache (because they have not been acknowledged by all matched *Reliable* DataReaders) can be removed after they expire. This is also applicable for *Transient Local* and *Reliable* DataWriters: samples that are in the DataWriter Cache because they need to be saved for future matched *Transient Local* and *Reliable* DataReaders may be removed after they expire.

[Note: Currently CoreDX DDS does not expire samples from a DataWriter's Data Cache based on Lifespan.]

Lifespan and the DataReader Cache

The Lifespan Qos policy setting is configured on the DataWriter by the publishing application, and is communicated to matched DataReaders. If the Lifespan *duration* is not INFINITE, a DataReader will remove samples from their Data Cache when they expire.

Filters (Time Based Filter, Content Filters)

CoreDX DDS provides the subscribing applications with options for filtering the data that is received by DataReaders.

The **Time Based Filter** QoS policy allows the application to indicate it does not necessarily want to see all data samples published for a Topic. In fact, the DataReader wants to see, for each instance, at most one data sample every time period. This time period is the *minimum_separation* for the Time Based Filter. Since the Time Based Filter operates on a per instance basis, the application of this filter will not affect the number of instances add to the DataReader Cache, only the number of samples that are added to the DataReader Cache.

The **Content Filtered Topic** is not a QoS policy, but a specialized kind of Topic where the subscribing application can apply a filter to its subscription. The filter is an SQL like statement. The ContentFilteredTopic is associated with another known Topic and applies a filter to the data available on that related topic. Since the Content Filter filters on the content of each data sample, it is possible that the filter can affect the number of samples and the number of instances created in the DataReader Cache.

In both of these filters, the DataReader filters the data before it is inserted into the DataReader Cache. That means only samples that pass through the filter will be added to the DataReader Cache. Filters are applied only to data samples – not to dispose and unregister commands.

When a sample is filtered, its associated instance is still created and added to the DataReader Cache (if not already there). This may result in instances in the DataReader cache that never have data samples associated with them. Since dispose and unregister samples always pass through filters, the states of these instances will be maintained appropriately, even if none of the samples associated with this instance are passed through the filter.

In Review: Adding to and Removing from the Data Caches

This section takes a step back from the details of CoreDX DDS QoS policies to look at the conditions where the Data Caches may grow (samples or instances are added to the Data Cache) or shrink (samples or instances are removed from the Data Cache).

DataWriter Cache

Samples are added to the DataWriter Cache under the following conditions:

- The publishing application creates a sample and there is room in its DataWriter Cache for the new sample, or if there is not room in its DataWriter Cache, the History QoS policy is configured for *KEEP_LAST*. Write, Unregister, and Dispose operations (DataWriter::write(), DataWriter::unregister_instance(), DataWriter::dispose(), and all variants of these operations) cause a sample to be created and added to the DataWriter Cache.

Samples are removed from the DataWriter Cache when:

1. The CoreDX DDS middleware in the publishing application has completed writing the sample. This happens when:
   - The CoreDX DDS middleware writes the sample to all *Best Effort* DataReaders AND
   - (Only if the DataWriter is *Reliable* and *Volatile*) The CoreDX DDS middleware has received an acknowledgement from all *Reliable* DataReaders

   ; or,
2. Samples expire based on the Lifespan *duration*; or,
3. A DataWriter has a History QoS Policy of *KEEP_LAST* and the cache already holds History *depth* samples and a new sample is created by write(), unregister() or dispose(); or,
4. A *Best Effort* DataWriter has non-INFINITE *max samples* or *max samples per instance* Resource Limits and the cache already holds the maximum samples and a new sample is created by write(), unregister() or dispose().

Instances are added to the DataWriter Cache when:

- The publishing application registers an instance that is not already registered. Every sample belongs to an instance, and the instance must be registered before a sample on that instance samples can be created. The application can explicitly register an instance by calling DataWriter::register_instance(), or CoreDX DDS will automatically register the instance when the application attempts to create a sample without first registering its associated instance.

Instances are removed from the DataWriter Cache when:

- The publishing application unregisters an instance. This must be done explicitly by calling DataWriter::unregister_instance().

DataReader Cache

Samples are added to the DataReader Cache when:

- A sample is received by the DataReader and the sample passes any filters configured on the DataReader and there is room in the DataReader Cache for the new sample and new instance (if the instance is not already in the DataReader Cache).
- If there is not room in the DataReader Cache for the sample, the new sample will be added only if:
  1. The instance that the sample belongs to already exists in the DataReader Cache or can be added to the DataReader Cache; and,
  2. the Reliability QoS policy is configured to *Best Effort* or the History QoS policy is configured to *KEEP_LAST* (in both cases, an older sample will be removed to make room for the new sample).

Samples are removed from the DataReader Cache when:

1. The subscribing application calls DataReader::take() (or one of take()'s variants); or,
2. Samples expire based on the source DataWriter Lifespan expiration *duration*; or,

3. A DataReader has a History QoS Policy of *KEEP_LAST* and the cache already holds History *depth* samples and a new sample is received; or,
4. A *Best Effort* DataReader has non-INFINITE *max samples* or *max samples per instance* Resource Limits and the cache already holds the maximum samples and a new sample is received; or,
5. A DataReader has non-INFINITE *autopurge nowriter samples delay*, and an instance state is determined to be NOT_ALIVE_NO_WRITERS; (associated samples will be removed after the specified delay); or,
6. A DataReader has non-INFINITE *autopurge disposed samples delay*, and an instance state is determined to be NOT_ALIVE_DISPOSED; (associated samples will be removed after the specified delay).

Instances are added to the DataReader Cache when:

- A sample belonging to the instance is received by the DataReader and the instance does not already exist in the DataReader Cache. The instance is created even if the associated sample is *not* added to the DataReader Cache due to filters or *max samples* or *max samples per instance* Resource Limits.

Instances are removed from the DataReader Cache when:

- The instance has a state of NOT_ALIVE_NO_WRITERS and there are no associated samples.

About Twin Oaks Computing

With corporate headquarters located in Castle Rock, Colorado, USA, Twin Oaks Computing is a company dedicated to developing and delivering quality software solutions. We leverage our technical experience and abilities to provide innovative and useful services in the domain of data communications. Founded in 2005, Twin Oaks Computing, Inc delivered the first version of CoreDX DDS in 2008. The next two years saw deliveries to over 100 customers around the world. We continue to provide world class support to these customers while ever expanding.

Contact

Twin Oaks Computing, Inc.
(720) 733-7906
+33 (0)9 62 23 72 20
755 Maleta Lane
Suite 203
Castle Rock, CO. 80108
www.twinoakscomputing.com