An Introduction to DDS and Data-Centric Communications

Gerardo Pardo-Castellote

Bert Farabaugh

Rick Warren

Abstract

To address the communication needs of distributed applications, a new specification is available called "Data Distribution Service for Real-Time Systems" (DDS). Many different types of distributed applications can use the DDS infrastructure as the backbone for their data communications.

This paper presents an overview of distributed applications, describes DDS specification and its components, and discusses how DDS can help developers design distributed applications.

Typical Distributed Applications

Many distributed applications exist today and many more are being planned for the future. One requirement common to all distributed applications is the need to pass data between different threads of execution. These threads may be on the same processor, or spread across different nodes. You may also have a combination: multiple nodes, with multiple threads or processes on each one.

Each of these nodes or processes is connected through a transport mechanism such as Ethernet, shared memory, VME bus backplane, or Infiniband. Basic protocols such as TCP/IP or higher level protocols such as HTTP can be used to provide standardized communication paths between each of the nodes. Shared memory (SM) access is typically used for processes running in the same node. It can also be used wherever common memory access is available.

Figure 1 shows an example of a simple distributed application. In this example, the embedded single board computer (SBC) is hardwired to a temperature sensor and connected to an Ethernet transport. It is responsible for gathering temperature sensor data at a specific rate. A workstation, also connected to the network, is responsible for displaying that data on a screen for an operator to view.

One mechanism that can be used to facilitate this data communication path is the *Data Distribution Service for Real Time Systems*, known as DDS.

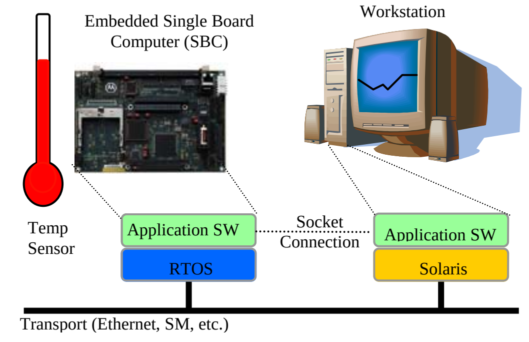

Figure 1. Simple Distributed Application

What is DDS?

The Data Distribution Service (DDS) specification standardizes the software application programming interface (API) by which a distributed application can use "Data-Centric Publish-Subscribe" (DCPS) as a communication mechanism. Since DDS is implemented as an "infrastructure" solution, it can be added as the communication interface for any software application.

Advantages of DDS:

- Based on a simple "publish-subscribe" communication paradigm
- Flexible and adaptable architecture that supports "auto-discovery" of new or stale endpoint applications
- Low overhead—can be used with high-performance systems
- Deterministic data delivery
- Dynamically scalable

- Efficient use of transport bandwidth
- Supports one-to-one, one-to-many, many-to-one, and many-to-many communications
- Large number of configuration parameters that give developers complete control of each message in the system

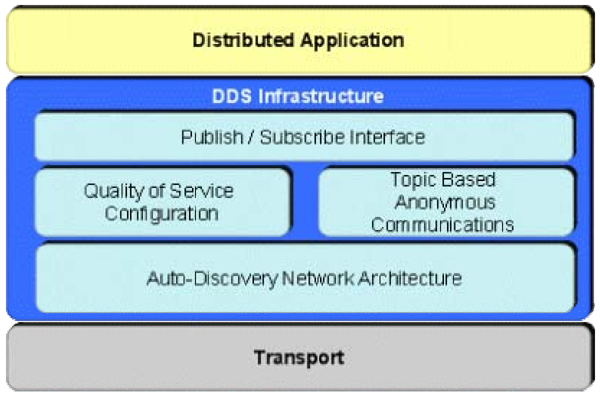
Distributed Application
DDS Infrastructure
Publish / Subscribe Interface
Quality of Service Configuration
Topic Based Anonymous Communications
Auto-Discovery Network Architecture
Transport

Figure 2. The DDS Infrastructure

As shown in Figure 2, DDS provides an infrastructure layer that enables many different types of applications to communicate with each other.

The DDS specification is governed by the Object Management Group (OMG), which is the same organization that governs the specifications for CORBA®, UML® and many other standards. A copy of the DDS specification can be obtained from the OMG website at www.omg.org.

What is "Publish-Subscribe"?

Publish-subscribe applications are typically distributed applications with endpoint nodes that communicate with each other by sending (publishing) data and receiving (subscribing) data anonymously. Usually the only property a publisher needs in order to communicate with a subscriber is the name and definition of the data. The publisher does not need any information about the subscribers, and vice versa.

As long as the interested applications know *what* data is being communicated, a publish-subscribe infrastructure is capable of delivering that data to the appropriate nodes— without having to set up individual connections. Publishers are responsible for gathering the appropriate data and sending it out to all registered subscribers. Subscribers are responsible for receiving data from the appropriate publishers and presenting the data to the interested user application.

What Does "Data-Centric" Mean?

Data-centric communication provides the ability to specify various parameters like the rate of publication, rate of subscription, how long the data is valid, and many others. These Quality of Service (QoS) parameters allow system designers to construct a distributed application based on the requirements for, and availability of, each specific piece of data.

A data-centric environment allows you to have a communication mechanism that is custom-tailored to your distributed application's specific requirements.

Solutions for using a "publish-subscribe" communication mechanism have typically been accomplished with proprietary solutions. The Data Distribution Service formalizes the data-centric publish-subscribe communication paradigm by providing a standardized interface.

In the example shown in Figure 1, DDS would be a software module on both the embedded SBC and the workstation. On the embedded SBC side, DDS would enable publishing of the temperature sensor data with specified QoS parameters. On the workstation side, DDS would enable a declared subscription to receive the temperature sensor data according to specified QoS parameters.

How Does DDS Help Developers?

By relying on a specification that governs the dissemination of data, distributed application developers can concentrate on the operation of their specific application—without worrying about how they are going to communicate with the other applications in the environment.

Applications that gather or generate data (through interfaces with sensors, files, on-board data computations, etc.) can use the DDS framework to send (publish) their data.

Similarly, applications that need data from other applications in a distributed environment can use the DDS framework to receive (subscribe to) specific data items. DDS handles all of the communications between publishers and subscribers.

By employing a "publish-subscribe" methodology for data communications, DDS provides abstract communications between data senders and receivers. Publishers of data are not required to know about each individual receiver, they only need to know about the specific data type that is being communicated. The same is true for subscribers. Subscribers do not need to know where the published data is coming from; they only need to know about the specific data type they wish to receive.

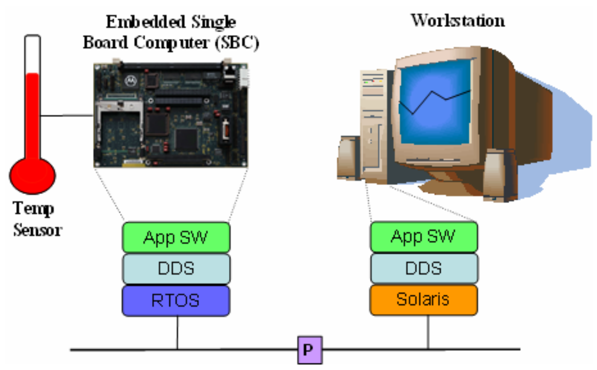

Figure 3. Simple Distributed Application w/ DDS

In Figure 3, the Embedded SBC publishes data packets with a simple "write" call using the current temperature sensor data as a parameter. On the workstation side, the application may either wait for data to become available or it can set up a callback routine that will receive the data immediately upon arrival. Once data is available, the workstation application can take the data from the DDS middleware.

An Overview of DDS Components

The specification for DDS is broken up into two distinct sections. The first section covers Data-Centric Publish-Subscribe (DCPS) and the second section covers the Data Local Reconstruction Layer (DLRL).

DCPS is the lower layer API that an application can use to communicate with other DDS-enabled applications. DLRL is the upper layer part of the specification that outlines how an application can interface with DCPS data fields through their own object-oriented programming classes. DLRL is an optional layer within the DDS specification.

For the purpose of understanding the basics and benefits of DDS, this paper will concentrate on the DCPS portion of the specification. DCPS is comprised of the following primary entities:

- Domain
- Domain Participant
- Data Writer
- Publisher
- Data Reader
- Subscriber
- Topic

Figure 4 shows how entities in DDS are related. The following sections contain more detailed descriptions of the entities used within DDS.

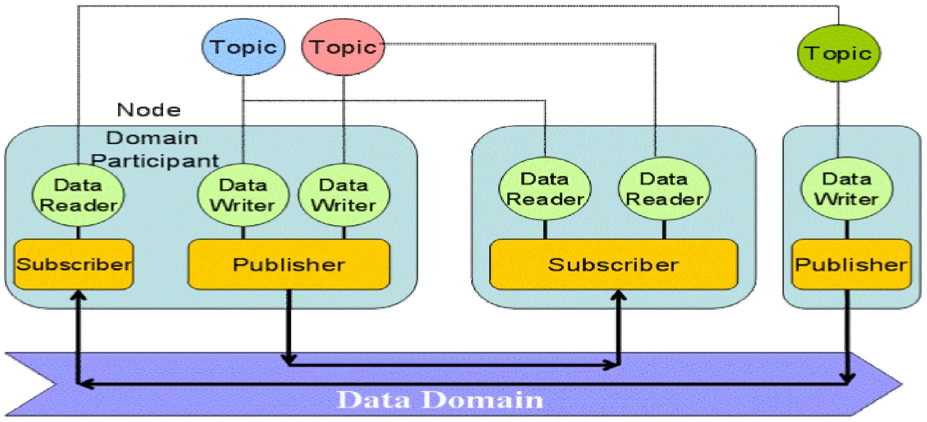

Figure 4. DDS Entities

Data is sent and received from the data domain. Publishers and Subscribers are used to manage single or multiple Data Writers and Data Readers, respectively. A Data Reader and Data Writer must be associated to the same Topic if data published by the Data Writer is to be received by the subscribing Data Reader.

Domains and Domain Participants

The domain is the basic construct used to bind individual applications together for communication. A distributed application can elect to use a single domain for all its data-centric communications.

Figure 5 shows an example of a system with six applications on five nodes, all communicating in the same domain.

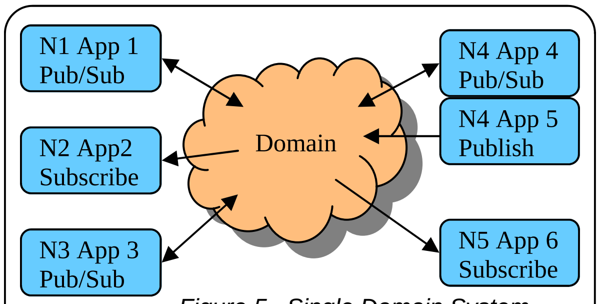

Figure 5. Single Domain System

All Data Writers and Data Readers with like data types will communicate within this domain.

DDS also has the capability to support multiple domains, thus providing developers a system that can scale with system needs or segregate based on different data types. When a specific data instance is published on one domain, it will not be received by subscribers residing on any other domains.

Multiple domains provide effective data isolation. One use case would be for a system to be designed whereby all Command/Control related data is exchanged via one domain while Status information is exchanged within another.

In Figure 6, three applications are communicating Command/Control data in Domain A and three other applications are communicating Status data in Domain B. For very large systems, developers may want to have different domains for each functional area in their overall system.

Multiple domains are also a good way to control the introduction of new functionality into an existing system. Suppose you have a distributed application that has been tested and validated as working correctly and then need to add new functionality. You want to minimize the impact of the new functionality and preserve the existing capability without breaking it.

In Figure 6, a new domain is added to the existing application. The new messages in Domain C will not affect the existing messages in the old domains. The new domain provides an isolated container for testing the new functionality. After testing is complete, the new applications (App 7 and App 8) can be added to the original system simply by changing their specified domain.

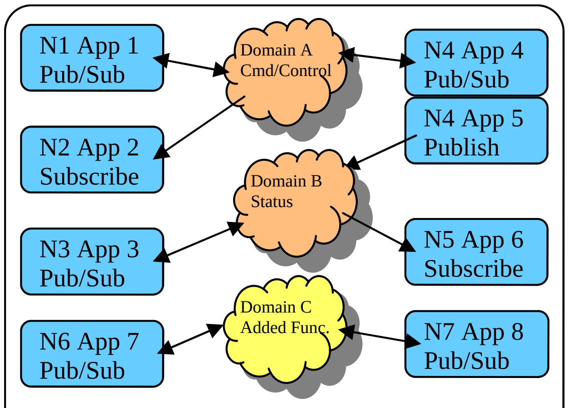

Figure 6. Multiple Domains System

An application uses an object called a “Domain Participant” to represent its activity within a domain. The Domain Participant object enables a developer to specify default QoS parameters for all data writers, data readers, publishers and subscribers in the corresponding domain. Default listener callback routines can be set up to handle events or error conditions that are reported back to the application by the DDS infrastructure. This makes it easy to specify default behavior that the application will follow when it receives a subscription for which it hasn’t set up specific listener routines for the underlying entities: Publisher, Subscriber, Data Writer or Data Reader.

Data Writers and Publishers

Data Writers are the primary access point for an application to publish data into a DDS data domain. Once created and configured with the correct QoS settings, an application only needs to perform a simple write call, such as in this C++ example:

```
retcode = writer->write(data, instance_handle);
```

The sending application controls the maximum rate at which data is published. Subscribers may have different requirements for how often they want to receive data. Some subscribers may want every individual sample of data, while others may want data at a much slower rate. This can be achieved by specifying a different time-based filter QoS for each subscriber. If a time-based filter is specified for an associated subscriber, then the publisher could be implemented to not send data to that subscriber any faster than is required. This would therefore reduce overall network bandwidth. When the write is executed, the DDS software will move the data from its current container, Data Writer, into a Publisher for sending out to the DDS Domain. Figure 7 shows how the entities (Domain Participant, Topic, Data Writer, and Publisher) needed to publish data are connected.

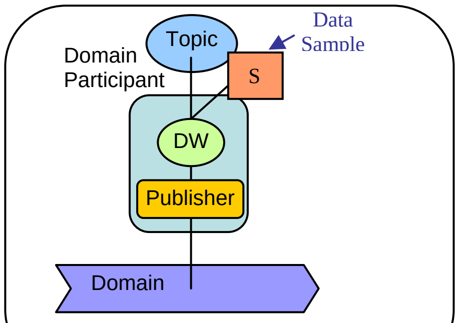

Fig. 7 Publication Model

The Publisher entity is just a container to group together individual Data Writers. A developer can specify default QoS behavior for a Publisher and have it apply to all the Data Writers in that Publisher's group.

Data Readers and Subscribers

A Data Reader is the primary access point for an application to access data that has been received by a Subscriber. Figure 8 shows the entities associated with subscriptions. Once created and configured with the correct QoS, an application can be notified that data is available in one of three ways:

- Listener Callback Routine
- Polling the Data Reader
- Conditions and WaitSets

The first method for accessing received data is to set up a listener callback routine that DDS will run immediately when data is received. You can execute your own specific software inside that callback routine to access the data.

The second method is to "poll" or query the Data Reader to determine if data is available.

The last method is to set up a "WaitSet", on which the application waits until a specified condition is met and then accesses the data from the Data Reader.

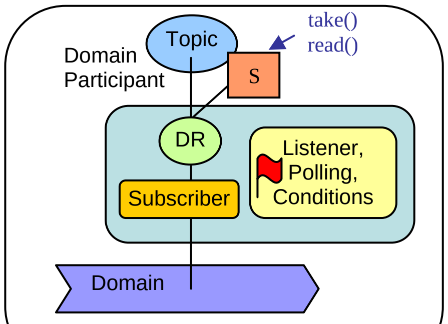

Fig. 8 Subscription Model

Having these three methods gives developers flexibility in accessing data. Accessing data is facilitated by calling take( ) or read( ) (the take() call removes the data from the middleware after returning it; the read() allows the same data to be retrieved multiple times). Just as Publishers are used to group together Data Writers, Subscribers are used to group together Data Readers. Again, this allows you to configure a default set of QoS parameters and event handling routines that will apply to all the Data Readers in that Subscriber's group.

Topics

Topics provide the basic connection point between publishers and subscribers. The Topic of a given publisher on one node must match the Topic of an associated subscriber on any other node. If the Topics do not match, communication will not take place.

A Topic is comprised of a Topic Name and a Topic Type. The Topic Name is a string that uniquely identifies the Topic within a domain. The Topic Type is the definition of the data contained within the Topic. Topics must be uniquely defined within any one particular domain. Two Topics with different Topic Names but the same Topic Type definition would be considered two different Topics within the DDS infrastructure.

The DDS specification suggests that types be defined in Interface Definition Language (IDL) files. IDL is the OMG standard language for defining object/data interfaces. The syntax for IDL is very similar to C++. The following primitive data types are supported by IDL:

- char
- octet
- short
- unsigned short
- long
- unsigned long
- long long
- unsigned long long
- float
- double
- long double
- boolean
- enum
- array and bounded sequence of the above types
- string

For example, here is a definition of a sample type found in a file named randomNumber.idl:

```
struct RndNum {
 float data;
 short processed;
 unsigned long seqNumber; //key
};
```

In this example, the type RndNum is a Topic Type. The Topic Name may be any string chosen by the application, such as "Random Numbers" or "Radar Samples".

Topic Keys

Within the definition of the Topic Type, one or more data elements can be chosen to be a "Key" for the type. The DDS middleware will use the Key to sort incoming data. By specifying one data element to be a Key, an application can then retrieve data from DDS that matches a specific key, or matches the next Key in a sequence of Keys. The container for holding one Key's worth of data is defined as an Instance.

Keys provide scalability. In the case of the distributed application shown in Figure 2, suppose there were multiple embedded SBCs, each with their own temperature sensor. Without Keys, you would need to create individual Topics for each of the different SBC/temperature sensor pairs. Topic names for these Topics might be:

- Temperature_Sensor_1
- Temperature_Sensor_2
- Temperature_Sensor_3
- And so on…

Therefore, even though each Topic is comprised of the same data type, there would still be multiple Topics. If you wanted to add another sensor into the environment, you would have to create a new Topic, Temperature_Sensor_N.

But with Keys, you would only need one Topic, named temperature, with the following Type definition:

```
struct Temperature {
 float data;
 unsigned long sensorId; //key
};
```

When a Subscriber receives data from all of the Publishers of "temperature," it will sort the data and present it to the application relative to its Key value, which in this case is the sensorId. New sensors could be added without creating a new Topic. The publishing application would just need to fill in the new sensorId when it was ready to publish that data.

An application can also have a situation where there are multiple publishers of the same Topic with the same Key defined. This enables the application to provide redundancy on a Topic. Redundancy is created by setting the QoS parameters Ownership and Ownership Strength. For a discussion on these QoS parameters, please refer to Appendix A.

MultiTopics & ContentFilteredTopics

In addition to standard Topics, there are also constructs in place for MultiTopics and ContentFilteredTopics.

A MultiTopic is a logical grouping of several Topics. It allows an application to select fields from multiple types and recombine them into a new type (something like an SQL SELECT statement). As a large application evolves, MultiTopics can be a powerful tool: the data types used within a subsystem may change, but the contents of those new types can be recombined to look like the old types, preserving the interface between subsystems.

A ContentFilteredTopic allows you to declare a filter expression by which only individual data samples of the specified Topic would be received and presented to the application. In our simple example, this would allow some subscribers to only receive and process data when a temperature exceeded a specific limit.

Quality of Service in DDS

The provision of QoS on a per-entity basis is a significant capability provided by DDS. Being able to specify different QoS parameters for each individual Topic, Reader or Writer gives developers a large palette from which to design their system. This is the essence of data centricity within DDS.

The DDS QoS parameters are:

- Deadline
- Destination Order
- Durability
- Entity Factory
- Group Data
- History
- Latency Budget
- Lifespan
- Liveliness
- Ownership
- Ownership Strength
- Partition
- Presentation
- Reader Data Lifecycle
- Reliability
- Resource Limits
- Time-Based Filter
- Topic Data
- Transport Priority
- User Data
- Writer Data Lifecycle

Appendix A contains a detailed discussion of what each QoS parameter is and how it is used to control communication. Through the combination of these parameters, a system architect can construct a distributed application to address an entire range of requirements, from simple communication patterns to complex data interactions.

Summary

The Data Distribution Service is a new OMG specification that creates a very simple architecture for data communication, while enabling very complex data patterns.

Topics allow endpoint nodes to be abstracted from each other, so nodes can enter and leave the distributed application dynamically. DDS is “data-centric”—all the QoS parameters can be changed on a per endpoint basis. This per endpoint configurability is the key to supporting complex data communication patterns.

DDS provides an API for sending and receiving data. It frees developers from having to worry about any network programming.

Simply put, DDS distributes data “where you want it, when you want it.” The publish-subscribe model makes it easy to specify the “where.” Data-centricity enables the “when.”

For additional information:

Dr. Gerardo Pardo-Castellote is chairman of the OMG Data Distribution Service Finalization Task Force and CTO of RTI in Santa Clara, CA. He can be reached at gerardo_pardo@omg.org. Bert Farabaugh and Richard Warren are members of the engineering staff at RTI. RTI created the first commercially available implementation of the DDS specification with its NDDS product line.

This paper derives from the OMG specification *Data Distribution Service for Real-Time Systems* available free of charge from www.omg.org.

Please contact us if you have additional questions or comments regarding this whitepaper:

Real-Time Innovations
3975 Freedom Circle
Santa Clara, CA 94089
http://www.rti.com
408.200-4700

This appendix presents detailed information on each of the QoS parameters found in DDS. Each section contains:

- QoS parameter name
- Entities to which the QoS applies
- Description
- Architectural benefits

The entities to which each QoS parameter applies are noted next to the parameter name, using the following abbreviations:

- DP = Domain Participant
- DR = Data Reader
- DW = Data Writer
- T = Topic
- Pub = Publisher
- Sub = Subscriber

Deadline (T, DR, DW)

Deadline indicates the minimum rate at which a Data Writer will send data. It also indicates how long a Data Reader is willing to wait for new data.

This policy is useful for cases where a Topic is expected to have each instance updated periodically. The default is a value of infinity; that is, there is no deadline.

For example, suppose an application requires a new "Position Command" to be received every 100 milliseconds. If that deadline passes without a new command, the application will be notified and can apply a default safe command until a new appropriate position command comes in.

Architectural Benefits:

- Allows each Data Writer to specify how fast it is able to publish data.
- Allows each Data Reader to specify how fast it needs to receive data.
- Indicates when a particular data source is not available, or that something may be wrong with a particular data source.
- Provides a way to set a time period during which a Data Reader will use the highest strength data source received when using multiple Data Writers.
- Allows an application to be notified immediately if a deadline is missed.

Destination Order (T, DR)

Destination Order determines how a Subscriber handles a situation in which it receives samples in a different order from that in which they were sent. The application can specify whether the samples should be read in the order in which they were sent or in the order in which they were received.

If a Data Reader receives multiple samples of the same instance, it needs some criteria for deciding how to logically order those sample values. Each sample includes two timestamps, a source timestamp and a destination timestamp. This parameter specifies which timestamp should be used for ordering the samples.

There are two possible settings:

- *By Source Timestamp*—Subscribers order the samples according to the source timestamp, which is attached to the data at its time and place of origin. This option ensures that all the Subscribers end up with the same final value for the instance. That is, all Subscribers will process the samples in the same order.
- *By Reception Timestamp*—Subscribers order the samples according to the destination timestamp, which is attached to the data when it reaches the Subscriber. Since the samples may reach each Subscriber at different times, the samples may be processed in different orders at each Subscriber node. Therefore, the various subscribers may end up processing the samples in different orders and thus end up with different final values. This is the default value.

Architectural Benefits:

- Allows an application to specify in what order it wants to receive the data from the DDS framework

Durability (T, DR, DW)

Durability specifies whether DDS will make past samples of data available to newly joining Data Readers in the system. Durability has three possible settings:

- *Volatile*—the system will not keep any past samples of data. This is the default setting.
- *Transient*—the system will keep a certain number of samples (determined by the Resource Limits and History parameters) in memory.
- *Persistent*—the system will keep all past samples on some type of non-volatile memory, such as a hard disk. This allows subscribers to join the system at any time and be able to receive the past N samples of publication data.

For example, if you want to save all publications from a temperature sensor, you could set the Data Reader's Durability to *Persistent*. Then each reading of the sensor would be saved off to disk.

Architectural Benefits:

- Enables new nodes that join an existing domain to receive past publication data
- Determines how past data shall be stored:
  - Not at all (volatile)
  - In memory (transient)
  - On disk (persistent)
- Persistent setting allows easy access to past messages for post-mission analysis

Entity Factory (DP, Pub, Sub)

All DDS entities must be "enabled" before they will become visible on the network and can be used for communication. The Entity Factory QoS policy is a mechanism whereby a user can indicate whether entities should be automatically enabled upon creation by their factory or whether enabling should be deferred to a later time of the user's own choosing. This parameter, when TRUE, and applied to a Domain Participant, would automatically enable all Publishers, Subscribers and Topics created by the Domain Participant factory. Also, when TRUE and applied to Publisher or Subscriber, would automatically enable Data Writers or Data Readers respectively when created. If this parameter were set to FALSE, all created entities of the factory would need to be enabled separately by the application before use. This parameter should help simplify overall code development by eliminating the need for specific enable functions to be called.

Architectural Benefits:

- Allows a developer to set a default enable upon creation of entities created under a factory.

Group Data (Publisher, Subscriber)

Group Data is extra information that is attached to the topic that may be accessed by the application when a new entity, Publisher or Subscriber, is discovered in the domain. Group Data is very similar to the User Data and Topic Data QoS parameters, but is intended to be applied on a per Publisher or Subscriber basis.

This QoS could be used to allow a system designer to apply application level policies that are similar to the capabilities that are provided by DDS with the Partition QoS parameter. One example would be that an application could further impose additional segmentation between topics by filtering on information contained within the Group Data QoS data field. While DDS would perform segmentation based on matching Partition strings, the application would use Group Data in a similar fashion. Just like in User Data or Topic Data (see descriptions below), this QoS parameter could be used for authentication or identification purposes as well. As show in figure 9, this publisher could contain an Identifier, Group Name or Signature that could then be accessed by listener routines associated with corresponding DataWriter or DataReader entities.

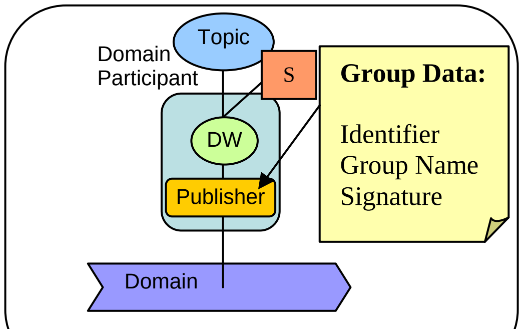

Fig. 9 Group Data Usage

Architectural Benefits:

- Associates system-level or architectural level information with the discovery of a new entity into the domain.
- Enables the use of authentication or signature data to be associated on a per Publisher or Subscriber basis.

History (T, DW, DR)

History specifies how many data samples will be stored for later delivery by the DDS infrastructure.

This parameter can be set to *Keep Last N*, where *N* is the number of data samples that will be kept or it can be set to *Keep All*. The default is *Keep Last N*, with *N* set to 1.

This parameter is important if an application has a dynamic architecture. When a new application joins the domain, subscribers to a Topic will receive the given number of past data samples. This is useful if the new application needs to "catch up" with what already happened with a particular Topic.

Architectural Benefits:

- Allows individual entities to specify their own degree of History. For example, one Data Writer may only want to keep its last 5 publications within the History queue, but another may want to keep 10 publications
- Allows historical data to be kept and accessed from both the Reader and Writer sides of a data sample transmission
- Allows system integrators to archive an entire mission's worth of data by setting the History QoS to *Keep All* and the Durability QoS to *Persistent*

Latency Budget (T, DR, DW)

Latency Budget is an optional guideline used by the DDS framework. It is used to implement optimization on the publishing side to accommodate subscribers' maximum acceptable delays (latency) in message delivery.

Architectural Benefits:

- Can be used in conjunction with a priority-based transport, where publications with low Latency Budget settings would get a higher priority when transmitted
- When working in two different environments, you can set this parameter to accommodate variable environment performance

Lifespan (T, DW)

Lifespan is the maximum duration that any one individual message is valid. Figure 10 shows a model of the Lifespan QoS parameter. When data is published by a given Data Writer, that data sample is only valid for the duration specified in the Lifespan parameter.

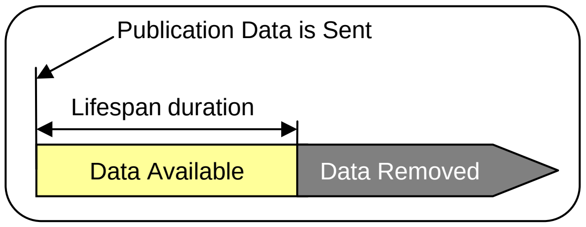

Fig. 10 Lifespan model

Then, at the time at which the Lifespan duration expires, data is removed from:

- Data Reader caches
- Any transient History queues
- Any persistent History queues

This also means that if a Data Reader has not accessed the data from its internal receive queue, then when the data is removed by the DDS infrastructure at the end of the Lifespan duration, data is not recoverable by the application. The data sample will be permanently removed by the DDS infrastructure. The default Lifespan duration is set to be infinite and therefore data is only removed by DDS when the developer specifies this parameter.

Architectural Benefits:

- Prevents the delivery of "stale" or old data to the application.
- Allows the DDS framework to remove stale data from the sending and receiving queues without required interaction by the user application.

Liveliness (T, DW, DR)

Liveliness specifies how the DDS infrastructure determines whether or not a DDS entity is still "alive." There are two sub-parameters in this QoS: the mechanism used to determine liveliness and a duration for how often a Data Writer commits to signaling that it is alive. The mechanism can be one of three values:

- *Automatic*—an "alive" signal is sent automatically at least as often as specified by the *duration*. This is the default setting. The default duration is infinity.
- *Manual by Participant*—the Entity is considered to be alive if any other Entities within the same Domain Participant are alive
- *Manual by Topic*—the Data Writer is considered to be alive only if at least one instance of its Topic has been asserted by the application

A Data Writer's liveliness is asserted when it writes data. It does not have to send a separate message to say it is alive.

If an entity does not have its liveliness status updated within the given period, it is determined to be inactive by the local application.

In Figure 9's example, the subscribing node would have to wait for the Command Module's Liveliness duration before determining that something had gone wrong and processing the Backup Module's command messages.

Architectural Benefits:

- Provides a way to tell if something is alive or stale
- Allows an application to change its behavior based on whether an entity is alive or not
- Allows applications in dynamically changing network environments to reconfigure themselves based on the liveliness of an entity
- Provides a way to have default or fail-safe commands to take over if primary entities do not assert their liveliness
- Allows applications to control their own assertions of liveliness or let DDS do it automatically

Ownership (T), Ownership Strength (DW)

The Ownership parameter specifies whether or not multiple Data Writers can update the same instance of a data object. When Ownership is set to *Shared* (the default), a data instance can be updated by multiple Data Writers. When set to *Exclusive*, a data instance can only be updated by one Data Writer. The selected Data Writer is determined by the Ownership Strength parameter. The combination of these two parameters provides the basis for implementing redundancy in the DDS messaging framework.

As shown in Figure 9, two modules, Command and Backup, are each issuing commands to the subscribing node. The Command module's Ownership Strength is 3, while the Backup module's is 2. In normal operation, since the Command module has a higher strength, the subscribing node only receives the "Grab Wafer" commands. However, if something goes wrong and the Command Module goes offline, the Backup Module becomes the highest-strength publisher and its "Go Home" command will be received by the subscribing node (assuming Ownership is set to *Shared*).

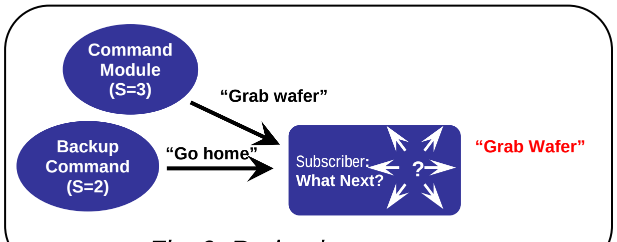

Fig. 9 Redundancy

Architectural Benefits:

- Specifies, on a per Topic basis, whether you want to receive all publications from all Data Writers, or to receive only the highest strength publication
- Provides instant fault tolerance if you have a primary and secondary publisher of the same Topic. If the something happens the to the primary publisher, the secondary will already be sending the appropriate data
- Allows dynamic changes to each individual publication strength, thereby changing the operation of a system while it is running

Partition (Pub, Sub)

Partition is a way to logically separate Topics within a Domain. The value is a string. If a Subscriber sets this string, then it will only receive messages from Publishers that have set the same string. (This is in addition to other requirements, such as matching Topics.) The default is an empty string, which means that no verification based on this parameter is performed.

Architectural Benefits:

- Provides a mechanism for isolating like-Topics within a single domain
- Allows an application to only accept data from a certain class of data sources (the partition can be used to designate each of the classes)
- Provides a way to achieve scalability within a domain
- Allows you to segment out certain partitions to achieve isolated unit testing

Presentation (Pub, Sub)

Presentation determines how data is presented to the application when dealing with a Subscriber that has a group of Data Readers associated with it. Presentation controls the extent to which changes to data-instances can be made dependent on each other in terms of their order, and how they are grouped together, if at all.

There are three sub-parameters contained within this one QoS:

- *Coherent Access* controls whether or not to keep groups of changes in a coherent set. This is turned off by default.
- *Ordered Access* controls whether or not to preserve the order of changes. This is turned off by default.
- *Access Scope* controls the scope used when considering how to group changes into coherent sets. You can set the granularity to one of three levels:
  - *Instance*—changes to separate instances are considered independent of each other. This is the default setting.
  - *Topic*—to be treated as part of a coherent set, the changes have to be from the same Data Writer.
  - *Group*—all coherent changes from all Data Writers attached to the same Publisher are grouped together.

When configured for Coherent Access, data will be presented to the application when the entire group of data has arrived. Otherwise, when configured for Ordered Access, data is presented in the order in which it sent by the Publisher.

Architectural Benefits:

- Allows applications to specify how data will be presented to a data reader:
  - Ordered access: By order of receipt
  - Coherent access: As a group based on a certain scope (per instance, per topic, or per group of topics)

Reader Data Lifecycle (DR)

Reader Data Lifecycle specifies a duration period in which a Data Reader will continue to hold data for Topic instances when no Data Writers for that instance exist any longer. The parameter for this policy is "autopurge_nowriter_sample_delay". If a given Data Reader instance of a Topic, as specified by its key value, does not have any more Data Writers sending data to it, then this parameter specifies how long to keep data in its receive queue before it is either taken by the application or removed by the DDS infrastructure. By default this parameter is set be unlimited so that the DDS infrastructure does not inadvertently remove data that wasn't specified for removal by the user application. If this was set to be unlimited and a particular data instance reaches a point where there are no Data Writers sending information to it, then data in this instance will just sit there, taking up memory resources of the Data Reader.

Architectural Benefits:

- Allows the developer to specify a time limit on freeing up resources for a given Data Reader.

Reliability (T, DW, DR)

Reliability specifies whether or not a given Data Reader will receive data "reliably" (no missed samples) from a Data Writer. DDS will take care of the retransmissions of data if so required.

There are two sub-parameters contained in this QoS: the kind of reliability desired (*Reliable* or *Best Effort*) and a duration (described below).

If the kind of reliability is set to *Best Effort* (the default), no data retransmissions will occur. The *Best Effort* setting is best suited for very cyclic data where the latest data sample is all that is needed. If all data samples of a given message are required to be received, this parameter should be set to *Reliable*.

The duration sub-parameter is only used in conjunction with the *Reliable* setting. The *Reliable* setting causes Data Writers to keep a certain number of sent messages in a queue, in case they need to be resent. Messages are removed from that queue after they have been acknowledged by the Data Readers. The duration specifies how long a Data Writer is allowed to block (wait) while waiting for space in the queue before it sends any more messages.

Architectural Benefits:

- Provides the primary mechanism for specifying the reliability of a single data topic or endpoint
- Allows different Data Readers to specify different reliability needs. As long as the Data Writer is offering *Reliable* service, each associated Data Reader can choose either *Reliable* or *Best Effort* delivery
- Provides reliable communications even on non-reliable transports

Resource Limits (T, DW, DR)

Resource Limits specify how much local memory can be used by the DDS infrastructure. This is very useful in platforms that use real-time operating systems where local memory use frequently comes at a premium.

You can specify the maximum number of instances, data samples, and data samples per instance. All of the parameters enable you to limit how much information is stored by DDS on a per message basis. The default for each limit is 1.

Architectural Benefits:

- Allows a system to limit the amount of resources that the infrastructure software will use
- Sets limits on:
    - Maximum instances per Topic
    - Maximum data samples per Topic
    - Maximum data samples per Instance
- Prevents an application from failing if too many other applications are sending data and using up all the resources of the local node
- Provides a failure indication if one publisher is sending data too fast and the local node has not implemented a Time-Based Filter

Time-Based Filter (DR)

Time-Based Filter provides a way to set a “minimum separation” period, which specifies that a Data Reader wants new messages no more often than this time period.

By default, this value is 0, which means that the Data Reader wants all values, regardless of how long it has been since the last one.

This parameter can be used by Data Readers to limit the number of data samples received for a given Topic. For example, suppose a publisher is sending out data every 100 milliseconds to multiple receivers, but not all the receivers need every sample of data. The Data Readers can use this parameter to scale down the periodicity of the publication, thereby saving unnecessary local CPU filtering.

Architectural Benefits:

- Prevents waste of CPU bandwidth for nodes that do not require all data samples from a Data Writer
- Allows some subscribing nodes to receive all data samples while others only receive every Nth sample
- Provides a throttle mechanism that is dictated by the subscribing side only

Topic Data (T)

Topic is extra information that is attached to the topic that may be accessed by the application when a new entity, Data Writer or Data Reader, is discovered in the domain. Topic Data is very similar to the User Data and Group Data QoS parameters, but is intended to be applied on a per Topic basis.

This QoS could be used to convey certain application data that is independent of the actual Topic Type. For example, if an application is started up into an existing DDS domain and that application instantiates a new subscription for the specified Topic, other entities that discover the new subscription will also be able to access its Topic Data. The Topic Data field could contain things like an application name or node name so that other applications could maintain a list of all applications or nodes that are accessing or providing that particular topic.

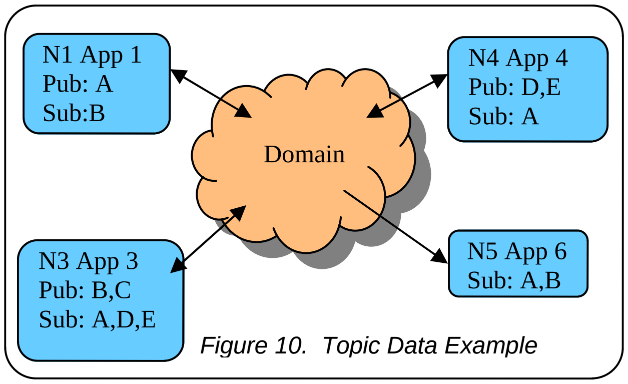

Figure 10. Topic Data Example

In the example given in Figure 10, if all the Topics were to include application and node information as part of the Topic Data, then when N4 joined the domain it would discover that for its subscription to Topic A, there exist these other following entities associated to Topic A:

- Publishers:
  - Node 1, Application 1
- Subscribers:
  - Node 3, Application 3
  - Node 5, Application 6

Architectural Benefits:

- Associates system-level or architectural level information with the discovery of a new entity into the domain.
- Enables the use of authentication or signature data to be associated on a per topic basis.

Transport Priority (T, DW)

Transport Priority is used to specify, to the underlying DDS infrastructure, how to set the relative priority of a given Topic message. This is only applicable for DDS implantations on top of transport mediums that support setting a priority.

Architectural Benefits:

- Allows a developer to assign a priority value to individual Topic messages in the domain.

User Data (T, DP, DR, DW)

User Data is extra information that is attached to the entity that instantiates it. User Data is very similar to the Topic Data and Group Data QoS parameters, but is intended to be applied on a per Data Writer and Data Reader basis.

This QoS could be used to set up authentication between Data Readers and Data Writers or applications. If the "signature" located within the User Data field does not match the approved signature list, then connections with those entities could be refused.

This QoS could be used in conjunction with these functions to prevent unwanted messages from appearing in the system:

- ignore_participant( )
- ignore_publication( )
- ignore_subscription( )
- ignore_topic( )

Architectural Benefits:

- Provides security or authentication information on a per Topic basis
- Associates system-level information with actual message data
- Implements publication/subscription permissions

Writer Data Lifecycle (DW)

Writer Data Lifecycle is a parameter that controls the automatic removal of data from a Data Writer's queue if it has recently been unregistered. For example, if there existed an application in which a Data Writer was responsible for reporting the location of several vehicles within a 100 mile radius, that Data Writer would contain a separate instance for each vehicle. If one of the vehicles were to leave the 100 mile radius, then the Data Writer would no longer need to publish the location of that vehicle. The parameter for this QoS policy is called "autodispose_unregistered_instances". If this parameter were TRUE, then when the application unregisters the instance for the vehicle in question, then all data samples for the instance will be removed from the history queue of the Data Writer. If the application does not unregister the vehicle that left the radius or if this parameter is set to FALSE, then the data associated with that vehicle will be held by the DDS infrastructure until the Lifespan QoS parameter has expired.

Architectural Benefits:

- Provides the application an auto removal of data for Topic instances that have recently been unregistered.

Appendix B: Glossary of terms used in this document

Callback Routine:	A Callback routine is application specific software that is executed directly by the infrastructure instead of the user's thread of execution.
Container:	A container is a logical grouping of items. Items can be nodes, applications, publications and subscriptions.
Entity:	An Entity in DDS refers to a basic architectural element that is to be used by developers. The primary DDS entities are: Publishers, Subscribers, Data Writers, Data Readers, Domain Participants and Topics.
Infrastructure:	An infrastructure is a base platform interface whereby all applications that are registered utilize a common set of capabilities.
Messages:	A message (otherwise known in DDS as a data sample) is an individual packet of data being delivered from a publisher to its intended subscribers.
Nodes:	A node is a computer or any electrical system that is equipped with a central processing unit. Typically, nodes in a distributed application communicate with each other over a transport such as Ethernet.
Processes:	A process is one or several threads of execution running together in the same address space to perform a certain functionality.
RTOS	Real Time Operating System: A real time operating system is an operating system that gives a developer greater control over the coordination of tasks running in an application.
Socket Connection:	A socket connection is a typical mechanism that is used to facilitate communications between two computing endpoints.
Stale Endpoint:	A Stale Endpoint is an application or computer that was once active within a distributed system, but now has been inactive for a specified period of time.
Threads of execution:	A thread of execution relates to one task or thread within an application whereby a set of commands are called in sequence.
Topic:	A Topic in DDS is the means by which data suppliers are paired with consumers. A Topic is comprised of a Topic name and a Topic type. The type is defined by the actual structure of data that is to be communicated.