WHITEPAPER

An Introduction to DDS and Data-Centric Communications

AUTHORS

Gerardo Pardo-Castellote
Chief Technology Officer, RTI

Bert Farabaugh
Director of Field Application Engineering, RTI

Andy Krassowski
Field Application Engineer, RTI

ABSTRACT

To address the communication needs of distributed applications, a family of trusted and proven specifications is available called Data Distribution Service (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 the core specifications of DDS and its components, and discusses how DDS can help developers design distributed applications and achieve data-centricity.

TYPICAL DISTRIBUTED APPLICATIONS

One requirement common to all distributed applications is the need to pass data between different independent modules or components. These modules may execute on the same processor, or spread across different nodes. It is common to have a combination — multiple nodes, with multiple processes on each one, each containing one or more modules.

Each of these nodes or processes is connected through a transport mechanism such as Ethernet, shared memory 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 (SHMEM) access is typically used for processes running in the same node.

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 real-time information exchange is DDS.

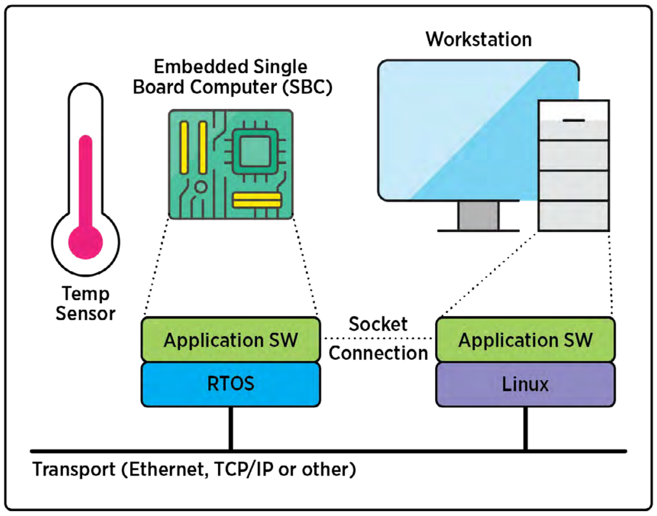

Figure 1. Simple Distributed Application

WHAT IS DDS?

DDS is a family of standards that specify the API, protocol, and security mechanisms that can be used by distributed applications to exchange real-time data. The software application programming interface (API) used by applications is based on a secure, Quality of Service (QoS)-aware "Data-Centric Publish-Subscribe" (DCPS) model. This means that applications need only be concerned with the Data that they wish to produce or consume, as well as the desired QoS. The DDS infrastructure takes care of the rest. Since DDS is implemented as an "infrastructure" solution, it can be added as the communication interface for any software application.

Advantages of DDS:

- Open standard and ecosystem. All aspects are governed by Object Management Group® (OMG®) standards, including the application APIs, Wire Protocol, Type System, Serialization formats and Security mechanisms. There is a rich ecosystem of vendors providing DDS-based products and applications.
- Based on a simple "Publish-Subscribe" communication paradigm. Supports one-to-one, one-to-many, many-to-one and many-to-many communications.
- Flexible and adaptable architecture that supports "auto-discovery" of applications and endpoints.

- Built-in mechanisms to monitor application and endpoint presence, availability and liveliness.
- Low overhead and serverless, so it can be used with high-performance systems.
- QoS-aware. Allows applications to specify non-functional characteristics of each dataflow, such as reliability, durability, lifespan, ownership, liveliness, etc. Developers can retain complete control of the individual dataflows in the system.
- Deterministic data delivery, so it can be used with real-time systems.
- Scalable, so it can handle large systems with thousands of applications and hundreds of thousands of individual data items.
- Efficient use of transport bandwidth.
- Fine-grained data-centric security. Every node that joins the system is authenticated, and fine-grained policies control what information each node can publish and subscribe. Dataflows are separately isolated and protected cryptographically.

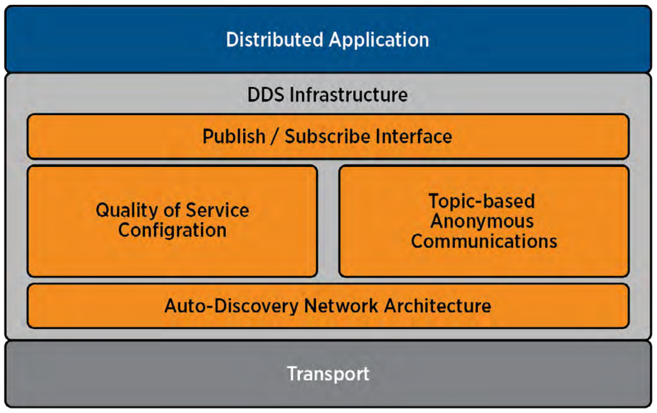
Distributed Application
DDS Infrastructure
Publish / Subscribe Interface
Quality of Service Configration
Topic-based Anonymous Communications
Auto-Discovery Network Architecture
Transport

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 specifications are governed by the OMG, which is the same organization that governs SysML, UML® and many other standards. A copy of any DDS specification can be obtained from the OMG website at www.omg.org/spec or www.dds-foundation.org/omg-dds-standard/. By formally specifying the on-the-wire data format, security mechanisms, Discovery protocol, APIs for multiple languages, and QoS behaviors, DDS enables developers to leverage proven techniques and future-proof their designs.

WHAT IS "PUBLISH-SUBSCRIBE"?

Publish-Subscribe (pub-sub) applications are typically built as a composition of modules that communicate with each other by sending (publishing) data and receiving (subscribing) data anonymously. Usually, the only thing a publisher needs in order to communicate with a subscriber is the ***Topic*** name (this identifies the flow) and the associated ***Data-Types*** (this defines the application-level APIs to read/write the data and the mechanisms to convert from application data to a network representation). The publisher does not need any information about the subscribers, and vice versa.

The pub-sub infrastructure is capable of delivering that data to the appropriate nodes—without having to manually set up individual connections. Publishers are responsible for gathering the appropriate data and sending it out to currently 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 refers to an architecture where data is the central aspect, while applications may come and go. In these systems, the data model precedes and outlives the implementation of any given application.

A data-centric communication infrastructure such as DDS provides the ability for distributed applications to "share state" instead of simply exchanging messages. Message exchange becomes a means to an end. With data-centric communications, it's possible to notify applications of relevant changes to the shared state, including changes to data values, the presence of new data-objects, and even the availability and liveliness of other applications accessing the shared state.

Data-centric communications can provide stronger consistency models to applications (e.g., eventual consistency), which greatly facilitates building robust and highly-available applications.

The data-centric model also provides a mechanism for applications to be decoupled in time. For example, a late-joining application can just observe the current state in order to "catch up" with the rest of the system. It does not need to process all the intermediate messages that caused the past state changes—rather, it only needs to focus on the end result.

A calendaring system provides a good example of shared state. Looking at the calendar gives you a snapshot of all the current appointments. Messages are sent to update the calendar (create/delete or change appointments), but the most important thing is the end result. Without support for accessing the (calendar) shared state, applications would have to process all the intermediate messages to arrive at the current state of the calendar.

Being data-centric, DDS provides APIs and protocols that go beyond message exchange and allow accessing and updating of the shared state.

WHAT DOES QoS-AWARE MEAN?

A QoS-aware communication infrastructure such as DDS provides the ability to specify various non-functional characteristics (Quality of Service parameters) regulating the exchange of information.

Functional characteristics of a system describe **what** the system should do—in other words, the functions it performs. Applied to the Publish-Subscribe infrastructure, it dictates the Topics it should publish and subscribe, the associated Data Types, when data-objects should be created and deleted, etc.

Non-functional characteristics of the system describe **how** the system should perform those functions. Applied to the Publish-Subscribe infrastructure, it dictates many characteristics of the information exchange, such as the use of various protocol-level mechanisms to ensure reliable data delivery, monitor application availability, ensure data freshness, control bandwidth and resource usage, etc.

DDS QoS parameters allow system designers to construct a distributed application based on the requirements for, and availability of, each specific piece of data. This can be used to optimize your distributed application to its specific requirements.

WHAT IS DATA-CENTRIC SECURITY?

Secure systems must protect the Confidentiality and the Integrity of the information, as well as provide Authentication and Access Control mechanisms to ensure data is only accessed by the intended applications.

DDS provides standard fine-grained security mechanisms that Authenticate each application prior to allowing it to join a DDS domain. Furthermore, fine-grained permissions control the Topics that each application can read and write, and built-in cryptographic mechanisms separately protect each dataflow end-to-end, ensuring the integrity and/or confidentiality of the data. A detailed description of the provided mechanisms is beyond the scope of this paper, but the details may be found in the DDS-Security™ specification available from the OMG (see www.omg.org/spec/DDS-SECURITY/).

HOW DOES DDS HELP DEVELOPERS?

Solutions for using a pub-sub communication mechanism have typically been accomplished with proprietary solutions. DDS formalizes the QoS-aware Data-Centric Publish-Subscribe communication paradigm by providing a standardized interface and the necessary protocols for achieving the required functionality.

In the example shown previously 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 delivery behaviors specified through QoS parameters. On the workstation side, DDS would enable a declared subscription to receive the temperature sensor data according to specified reception characteristics defined by QoS parameters.

By relying on a specification that governs the dissemination of data and also provides access to shared state, distributed application developers can concentrate on the operation of their specific modules—without worrying about how they are going to communicate with the other modules in the system.

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 system can use the DDS framework to receive (subscribe to) specific data items. DDS handles all of the communications between publishers and subscribers.

By employing the pub-sub methodology for data communications, DDS abstracts communications between data senders and receivers. Publishers are not required to know about each individual receiver—they only need to know about the specific Topics that are being shared and how to send it. 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 and how to receive it.

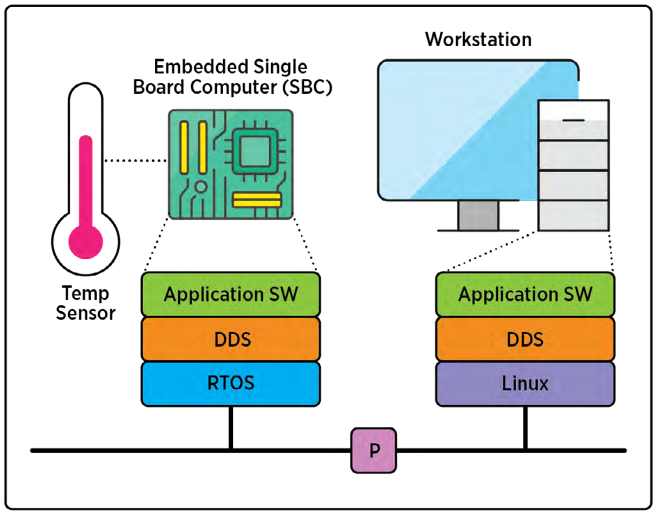

Figure 3. Simple Distributed Application with DDS

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 block while waiting for data to be available, set up a callback routine to be notified immediately when data arrives, or check its local cache on-demand to access the latest value.

AN OVERVIEW OF DDS ENTITIES

The DDS API defines the following entities:

- DomainParticipant
- DataWriter
- Publisher
- DataReader
- Subscriber
- Topic

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

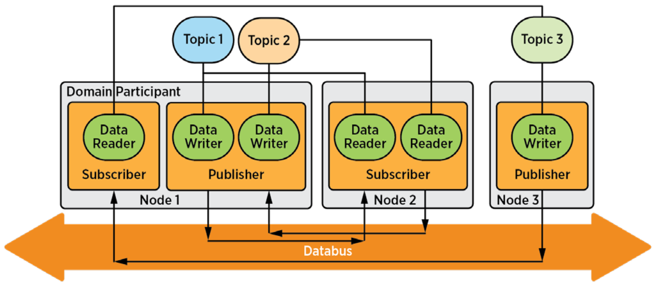

Figure 4. DDS Entities

In Figure 4, DomainParticipants join a DDS Domain. The domain represents a shared data-space (databus) that provides access to the shared state. The shared state is organized into named Topics. Each Topic may contain multiple data-objects. Each data-object is identified by the value of the designated key-fields within the object. Data is sent and the data-objects are updated using a DataWriter. Each DataWriter is associated with a single Topic so it can only be used to send or update data-objects belonging to that Topic. Likewise, the data-objects are accessed using a DataReader. Each DataReader is associated with a single Topic and can only access and receive data-objects belonging to that Topic. Publishers are used to group and manage DataWriters. Subscribers group and manage DataReaders.

DOMAINS AND DOMAINPARTICIPANTS

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.

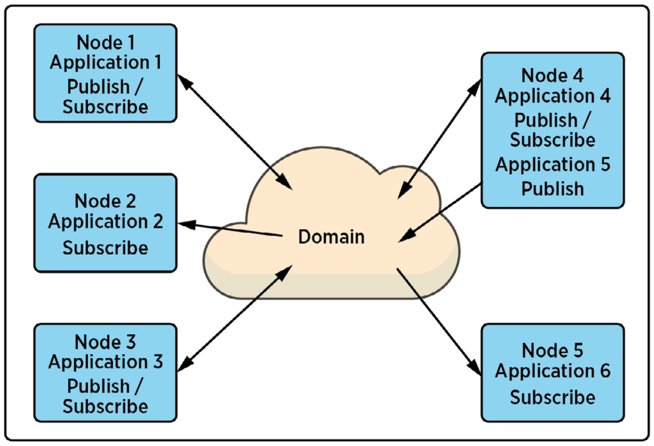

Figure 5. System with Single Domain

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 access needs. When a specific data instance is published on one domain, it will not be received by Participants attached to any other domains.

Multiple domains provide effective data isolation. One use case would be for a system to be designed wherein 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 dataflows in Domain C will not affect the existing dataflows 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.

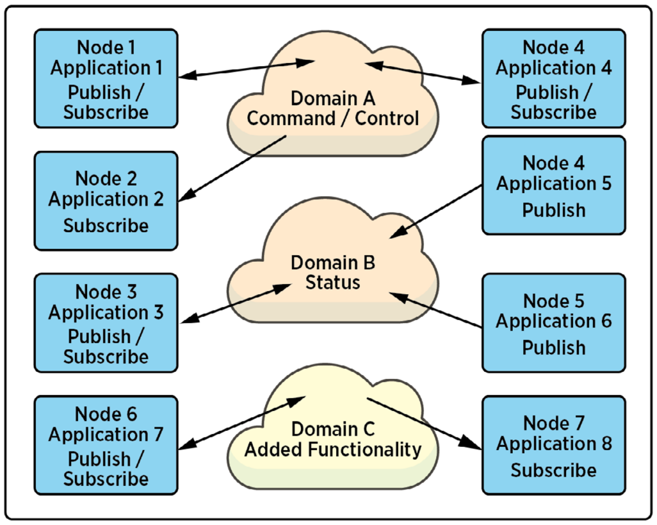

Figure 6. System with Multiple Domains

An application uses a "DomainParticipant" to join a DDS Domain. The DomainParticipant 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 the default behavior that an application will follow when it receives notifications for which it hasn't set up specific listener routines on the associated entities: Publisher, Subscriber, Data Writer or Data Reader.

A DDS domain is identified by two parameters: An integer **domain ID** and a string **domain Tag**. These are specified when each DomainParticipant is created. Only participants that specify the same values for both parameters (ID and Tag) will join the same Domain and thus be able to discover and communicate with each other.

DATAWRITERS AND PUBLISHERS

DataWriters are the primary access point for an application to publish data into a DDS 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:

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

The sending application controls the 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. Some Subscribers may want to see every update to an object, while others are more interested in the latest update (i.e., the current state). This can be achieved by specifying different QoS policies, such as HISTORY or TIME_BASED_FILTER. If a time-based filter is specified for a DataReader, then the DataWriter can avoid sending data updates to that reader any faster than it actually requires. This would therefore reduce overall network bandwidth. When the write is executed, the DDS software will copy the data from the local Data Writer cache into the local caches of each of the DataReaders for the Topic. Figure 7 shows how the entities (DomainParticipant, Topic, DataWriter and Publisher) needed to publish data are related.

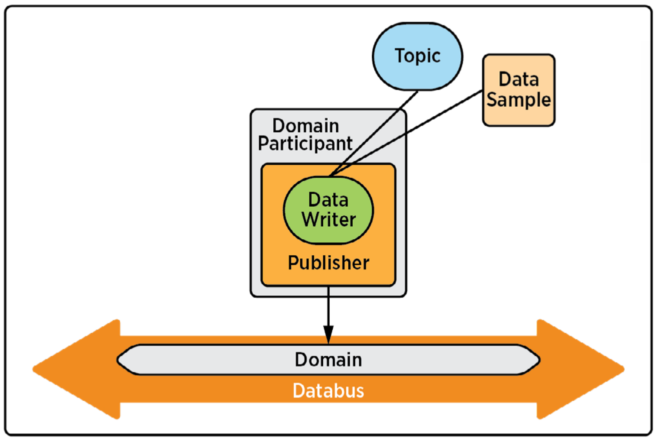

Figure 7. Publication Model

Publishers are used to group together individual DataWriters. A developer can specify default QoS behavior for a Publisher and have it apply to all the DataWriters in that Publisher's group.

DATAREADERS AND SUBSCRIBERS

A DataReader is the primary access point for an application to access data in the DDS domain. 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 DataReader
- Waiting on a WaitSet until specific Conditions trigger

The first method for accessing received data is to set up a listener callback routine that DDS will call 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 local DataReader cache to determine if data is available.

The last method is to set up a "WaitSet", on which the application waits until a specified set of conditions are met and then accesses the data from the DataReader.

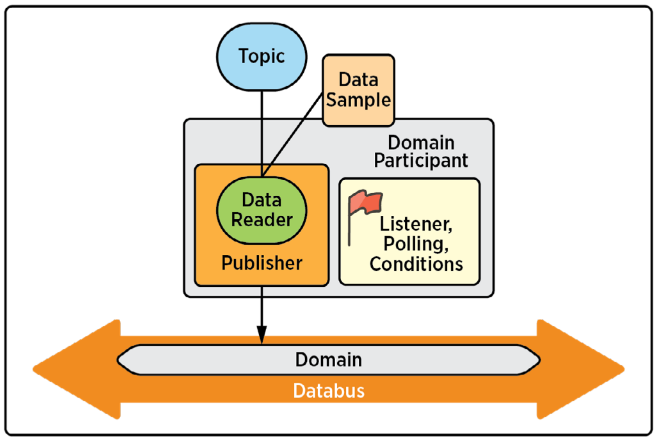

Figure 8. Subscription Model

Having these three methods gives developers flexibility in accessing data. Accessing data is accomplished by calling `take()` or `read()` on the DataReader. The `take()` operation removes the data from the local DataReader cache after the application uses it; the `read()` operation leaves the data on the local DataReader cache after the application uses it, allowing the application to access the data multiple times.

Subscribers are used to group together individual DataReaders. Similar to Publishers before, this allows you to configure a default set of QoS parameters and event handling routines that will apply to all the DataReaders in that Subscriber's group.

TOPICS

Topics provide the logical connection between DataWriters, DataReaders and the shared global data-space (Domain).

In the shared data-space, each topic is identified by a **Topic Name** which is a simple string. The name uniquely identifies the Topic within the DDS Domain.

If you are familiar with Publish-Subscribe systems, you may think of each Topic as identifying a separate information flow. From the data-centric perspective, each Topic represents a separate collection of data-objects. All data-objects in a Topic either have the same Type or closely-related (compatible) types. Moreover, each data-object is identified by the values of certain fields in the data-object that have been designated as the "key fields".

Topics in the shared data-space will group related objects whose state is observed collectively. For example, an air-traffic management application may use a Topic called "FlightStatus" to share the current location and status of all flights. Each individual flight could be identified by a combination for key fields (e.g., Airline and Flight Number), or some other unique identifier (e.g., the aircraft Tail Number) could also be used as the key. Grouping all these objects into a single "FlightStatus" Topic allows a single DataReader to observe (subscribe) to the status of multiple flights, be notified of the creation/appearance and completion of flights, etc. Likewise, a single DataWriter can be used to update multiple flights.

For more details on the use of keys, see below under "Topic Keys."

Each DomainParticipant must create local Topic objects to identify the shared data-space topics it wants to access (read or write). When creating the local Topic object, the application must specify both the Topic name and an associated data-type.

To publish data, the application creates a DataWriter object and associates it with a local Topic. Likewise, to subscribe to data, the application creates a DataReader object and associates it with a local Topic. For communication to occur, the Topic name associated with the DataWriter must match the Topic name associated with the DataReader. The Data Type, associated with the DataWriter and DataReader topics, however, need not be the same. They just need to be compatible. We will elaborate on this further below.

DATA TYPES

In data-centric systems, the data that is published and subscribed always has an associated Data Type. The DataWriters and DataReaders are specific to the published/subscribed type and deliver strongly-typed data to the application.

However, it is not required for the Data Types to be known at compile time, because DDS provides both static and dynamic APIs. For example, dynamic APIs enable applications to create writers and readers of arbitrary types, with types defined or discovered at run-time. Nevertheless, once a DataWriter or DataReader is created, its type is fixed, even if accessed using a dynamic API.

Strongly-typed systems provide stronger contracts, which facilitates designing and composing modular systems. They also provide more deterministic behavior and can leverage language mechanisms for better performance and resource management. These are all important aspects of real-time, dependable, safe, critical-infrastructure systems, which form the usual application space for DDS.

However, unless care is taken, distributed strongly-typed systems can also be rigid and inflexible, preventing system composability and evolution.

For this reason, DDS does not require that DataWriters and DataReaders are associated with identical Data Types. Rather, DDS only requires that the types are compatible, or more precisely, that the type published by the DataWriter is assignable to the one that is subscribed to by the DataReader. The precise assignability rules, defined in the DDS-XTypes™ specification, are beyond the scope of this paper. In general, the type assignability rules are defined to allow safe type evolution – that is, the type of the DataWriter can differ from that of the DataReaders, as long as it is possible for the reader to understand the information it expects and not miss any "essential" information needed to properly interpret the data.

The DDS specifications provide various mechanisms to define Data Types. The most popular approach is to use the OMG Interface Definition Language version 4.x (IDL4). IDL4 is both an ISO and OMG standard language for defining Data Types and interfaces. The syntax for IDL4 is very similar to C++.

The following primitive Data Types are supported by IDL4:

- char, wchar, octet
- int8, uint8
- int16, short, uint16, unsigned short
- int32, long, uint32, unsigned long
- int64, long long, uint64, unsigned long long
- float, double, long double
- boolean
- enum
- bitmap, bitset
- string, wstring

For example, here is a definition of a Data Type used in the popular RTI "shapes demo:"

```
@appendable
struct ShapeType {
  @key string<256> color;
  @range(0, 256) int16 x;
  @range(0, 256) int16 y;
  @min(10) uint16 size;
};
```

In this example, the type ShapeType is a Topic Type. The Topic Name may be any string chosen by the application, such as, "Square" or "Circle." In addition, this example shows some IDL keywords that are used to reflect various aspects of the fields. The "@key" references the field(s) that will comprise the Topic Key (described below) and the "@range" reference is to make note of the fact that the values for x and y in this case can only be between 0 and 256. A full list of IDL keywords can be found in the OMG IDL specification.

TOPIC KEYS

Within the definition of the Topic Type, one or more data members can be chosen to be a "Key" for the type. DDS uses the value of the Key members to identify each separate data-object within the Topic. Updates to Topic data-objects with different keys are independent from each other. In other words, an update of a Topic data-object identified by Key value **K1** modifies the values for the remaining members in the **K1** object, but does not affect or replace the corresponding values of other data-objects in the Topic. Using the "shapes" example above, a data-object for Topic "Square" where the ShapeType member **color** has value "Blue" updates is different from a data-object in the same Topic "Square" where the **color** is "Green." Hence, an application can update the **x**, **y** and **size** values of the "Blue" data-object within Topic "Square," without impacting any other data-objects on that same Topic (e.g., a "Green" data-object in Topic "Square").

DDS can retain a certain history of the updates made to each data-object providing APIs to read the current and most recent updates for each data-object (by specifying the Key).

DDS can also notify the application of the appearance and/or disappearance of data-objects. For example, the reception of data on Topic "Square" with a color value that was not seen before, or the disappearance from the system of all the DataWriters that were updating a particular data-object (e.g., the "Blue" data-object in Topic "Square").

Keys may also be used to achieve finer control over dataflows. Each separate Key value creates an independent information flow and DataReaders can choose to only read data-objects that have certain values of the Keys.

The use of Keys is also important when building scalable applications. Having a Topic with many keys is far more resource-efficient than having many Topics (e.g., one per Key value). It can also simplify application development and evolution. In the case of the distributed application shown earlier in Figure 3, 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...

Despite each Topic having the same Data Type, the fact that we are using different Topic names would force the creation of multiple Topics and, correspondingly, multiple DataReaders and DataWriters to read/write the separate Topics. If you wanted to add another sensor into the system, you would have to: create a new Topic, "Temperature_Sensor_N"; create a new DataWriter(s) to write it; and create a new DataReader(s) to read it.

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

```
struct TemperatureType {
  @key uint32 sensorId;
  float value;
};
```

When a Subscriber receives data from all of the Publishers of "Temperature," it will 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.

Keys facilitate many-to-one scenarios in which multiple applications write data to the same Topic. As long as the applications write separate keys, each flow can be kept separately — and the values of one DataWriter do not conflict with those written by the others. It is also possible for multiple DataWriters to update the same data-object (i.e., write updates that reference the same value for the Key members). This can be used to provide redundancy and/or arbitrate the ownership/control of specific data-objects. Redundancy and ownership control scenarios are configured using the QoS policies Ownership and Ownership Strength.

CONTENT FILTERED TOPICS

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

A ContentFilteredTopic allows a DataReader to specify a filter expression that selects only a subset of the data samples published on the specified Topic to be received and presented to the subscribing application. In our simple example, this could allow some subscribers to only receive and process data when a temperature exceeded a specific limit of interest.

By leveraging the standardized solutions to common design issues, DDS users are able to focus on their application logic and let the communications framework take care of the data movement as specified.

QUALITY OF SERVICE IN DDS

The provision of QoS policies for each DDS Entity is a significant capability provided by DDS. Being able to specify different QoS policies for each individual DomainParticipant, Topic, Publisher, Subscriber, DataReader or DataWriter gives developers a large palette from which to design their system. This is the essence of the QoS-aware data-centricity provided by DDS.

The DDS QoS parameters include:

- 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

Through the combination of these policies, 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 standard is a family of OMG specifications that creates a simple yet powerful architecture for information exchange and application integration.

Topics allow nodes and application modules to be abstracted from each other, so nodes and modular components can enter and leave the distributed application dynamically. DDS provides a QoS-aware "data-centric" Publish-Subscribe model, so QoS policies can be configured on a per-endpoint basis. This fine-grained configurability is critical to supporting complex data communication patterns.

DDS provides a standards-based API for sending and receiving data in an expanding list of languages (C, C++, Java, C#, Ada and Python). It provides the means to define data-models, and a standard wire protocol to ensure interoperability between modular components developed by different vendors. It frees developers from having to worry about any network programming, data serialization and representational differences by using a single data-model.

Simply put, DDS distributes data "where you want it, when you want it, how you want it." The Publish-Subscribe model makes it easy to specify the "where." Data-centricity enables the "when," and QoS-awareness enables the "how."

FOR ADDITIONAL INFORMATION:

Dr. Gerardo Pardo-Castellote is Co-Chair Data Distribution Service SIG for OMG and CTO of RTI in Sunnyvale, CA. He is the primary author of many of the DDS specifications. Bert Farabaugh and Andy Krassowski are members of the engineering staff at RTI. RTI created the first commercially available implementation of the DDS specification in its product line.

This paper is derived from the OMG specification Data Distribution Service for Real-Time Systems, available free of charge from www.omg.org. Please contact us at info@rti.com if you have questions or comments regarding this whitepaper.

APPENDIX: GLOSSARY OF TERMS USED IN THIS DOCUMENT

Callback Routine: A Callback routine is an application-provided function that is called by the DDS infrastructure in order to notify the application of relevant events. Callbacks are typically called in the context of the DDS infrastructure threads, but there are also mechanisms where applications can provide the thread pools that execute these calls.

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 class of Software Objects that form the foundation of the DDS API. These objects allow applications to control the operation of the DDS infrastructure, join DDS Domains, define what is being pushed and subscribed, access data, etc. The primary DDS entities are: Publishers, Subscribers, DataWriters, DataReaders, DomainParticipants and Topics.

Infrastructure: An infrastructure is a base platform interface whereby all applications that are registered utilize a common set of capabilities.

Messages (Data Samples): A message (otherwise known in DDS as a Data Sample) is an individual packet of information being delivered from a publisher to its intended subscribers. Data Samples contain application-level data, as well as information about the data (e.g., the timestamp, sequence number, identification of the sender, etc.).

Nodes: A node is a computer element that is connected to other nodes forming a network. Typically, nodes are distributed and communicate with each other over a transport, such as Ethernet.

Processes: Processes refers to a computer program composed of one or more threads of execution running in the same address space to perform a certain functionality.

RTOS: A Real-Time Operating System (RTOS) is an operating system that provides deterministic timing and resource usage behavior, such that applications are able to better predict their timing behavior and consistently meet real-time constraints.

Socket: A socket is a typical mechanism provided by Operating Systems that can be used to communicate between processes which may reside in the same or different Nodes.

Topic: A Topic in DDS is the means by which data producers are paired with consumers. In the shared data-space, the Topic is defined by its string Topic name. Locally, each application defines the Topics it uses by providing the Topic name and the Data Type they intend to use with the Topics. The Data Types used by different applications to communicate on a Topic need not be identical. It is sufficient for types to be compatible. This allows applications to evolve and modify the Data Types without breaking interoperability.

ABOUT RTI

Real-Time Innovations (RTI) is the largest software framework company for autonomous systems. RTI Connext® is the world's leading architecture for developing intelligent distributed systems. Uniquely, Connext shares data directly, connecting AI algorithms to real-time networks of devices to build autonomous systems.

RTI is the best in the world at ensuring our customers' success in deploying production systems. With over 1,800 designs, RTI software runs over 250 autonomous vehicle programs, controls the largest power plants in North America, coordinates combat management on U.S. Navy ships, drives a new generation of medical robotics, enables flying cars, and provides 24/7 intelligence for hospital and emergency medicine. RTI runs a smarter world.

RTI is the leading vendor of products compliant with the Object Management Group® (OMG®) Data Distribution Service (DDS™) standard. RTI is privately held and headquartered in Sunnyvale, California with regional offices in Colorado, Spain and Singapore.

Download a free 30-day trial of the latest, fully-functional Connext software today: www.rti.com/downloads.

Your systems.
Working as one.

CORPORATE HEADQUARTERS
232 E. Java Drive, Sunnyvale, CA 94089
Telephone: +1 (408) 990-7400
Fax: +1 (408) 990-7402
info@rti.com

rti.com
rti_software
rtisoftware
company/rti
connextpodcast
rti_software