OMG Data Distribution Service: Real-Time Publish/Subscribe Becomes a Standard

The recently adopted Data Distribution Service standard from the Object Management Group brings publish-subscribe data distribution middleware technology to the realm of networked real-time applications.

by Gerardo Pardo-Castellote
Real-Time Innovations, and Chairman, DDS Standards Committee

In June 2004, the Object Management Group finalized the Data Distribution Service (DDS) Specification for Real-Time Systems. This is the most significant addition to the portfolio of specifications addressing the needs of real-time systems that OMG has made in recent years. Despite the novelty, the technology is well proven. The DDS standard unifies some of the best practices present in successfully deployed real-time data distribution middleware such as NDDS from Real-Time Innovations and Splice from Thales.

DDS enables applications to use a much simpler programming model when dealing with distributed data-centric applications. Rather than developing custom event/messaging schemes or artificially creating wrapper CORBA objects to access data remotely, the application can identify the data it wishes to read and write using a simple topic name, and use a data-centric API to directly read and write the data.

As illustrated in Figure 1, DDS creates the illusion of a shared "global data space" populated by data objects that applications in distributed nodes can access via simple read and write operations. In reality, the data does not really "live" in any one computer's address space. Rather, it lives in the local caches of all the applications that have an interest in it. Here is where the publish/subscribe aspect becomes key.

Publish-Subscribe and Data Distribution

The classic distributed shared memory model allows applications to access data remotely using simple read and write operations. However, these architectures don't scale beyond SMP computers or tightly coupled clusters. The reason is that the random access semantics of memory and the implied totally reliable, "instan-

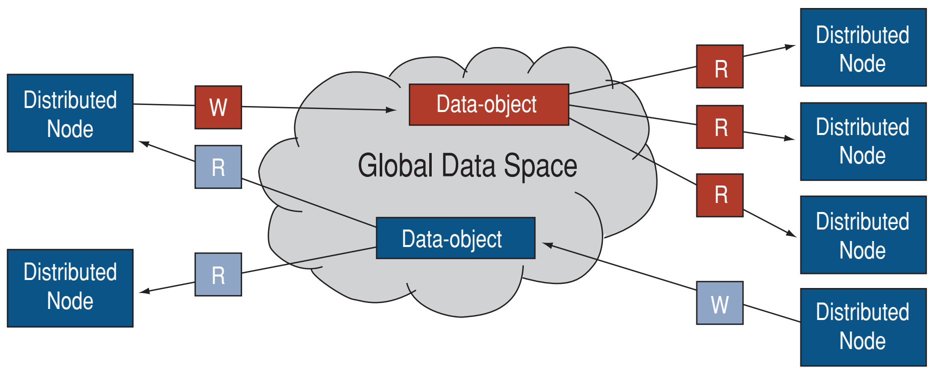

Figure 1 Overall DCPS model. Distributed nodes are able to read (R) and write (W) data objects that live in a shared virtual global data space.

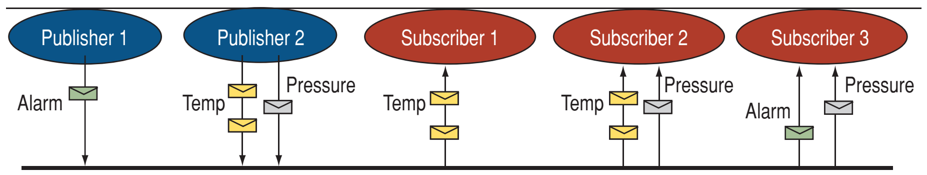

Figure 2 Topic-based publish/subscribe. Topic is an application-selected label—such as “pressure” or “temp”—used to identify publications and subscriptions.

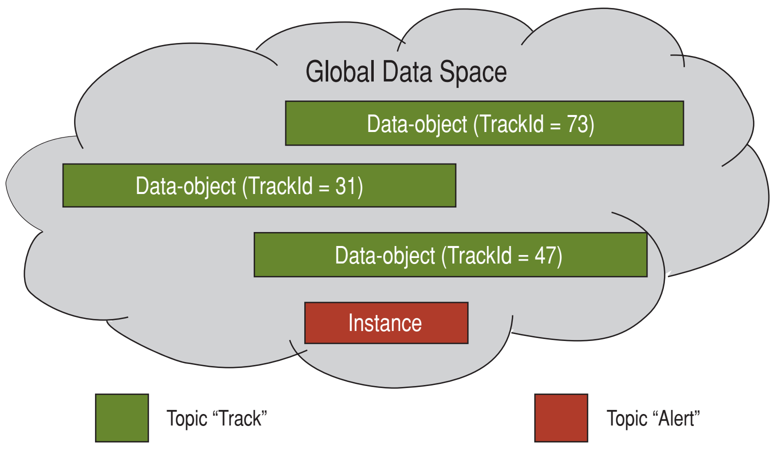
Global Data Space
Data-object (TrackId = 73)
Data-object (TrackId = 31)
Data-object (TrackId = 47)
Instance
Topic “Track”
Topic “Alert”

Figure 3 Topic and key identify data objects. A data object is identified by the combination of a topic (such as “Track”) and a topic-specific key (such as a TrackId). Some topics (e.g., “Alert” above) may have no key indicating the reads/writes apply to a single data stream.

taneous” response cannot be implemented transparently in a LAN or WAN where computers can join and leave and communication links can have sporadic faults. Hiding all these details from the application is not practical—the model is simply not a good fit for the physical realities of a distributed system.

The publish/subscribe model has steadily gained popularity in the context of programming distributed systems. The key concept behind publish-subscribe is very simple. Applications must be programmed in such a way that the “declaration of information access intent”—that is, what the application wants to do—is separate from the information exchange itself. This separation allows the middleware to prepare itself by reserving all the needed resources such that the information access can be as efficient as possible.

In the context of data distribution the publish/subscribe approach means that the application must declare its intent of writing data and specify which data objects it wishes to write (i.e., define its publications) and similarly it must declare its intent to read data and specify which data objects it intends to read (i.e., define its subscriptions) before it actually writes or reads the data itself.

A key aspect of any publish/subscribe system is how the applications identify what they intend to publish or subscribe to. As shown in Figure 2, most publish/subscribe systems use a combination of an application-selected label (called a “topic”) and a filter on the content of the data itself. DDS uses a slightly more powerful approach that combines topic, content and a special field (the key), which identify each data object in the global data space.

The DDS specification was developed with the needs of real-time systems in mind. To this end, it defines a comprehensive list of quality of service (QoS) parameters that allow the application to finely tune the resources used by the system. In addition, the API includes the programmable call-back mechanism designed to provide very high performance in terms of latency and throughput.

The basic concept of publish/subscribe has been applied to things beyond data distribution such as event and message distribution. These other middleware technologies differ in their information exchange model, which is message-centric instead of data-centric. In other words, the application cannot think in terms of being able to read or write data. Rather it is restricted to viewing the system in terms of sending messages (or events) to other applications. In practical terms, this means that the data model will have to be built by the user in a custom way at considerable development cost. On the other hand, the DDS model supports event propagation and messaging by means of special QoS settings. In that sense, the Data Distribution model subsumes the capabilities of many event distribution and messaging models.

In data-centric systems, the information exchanges refer to values of an imaginary global data object. Given that new values typically override prior values, both application and middleware need to identify the actual instance of the global data object to which the value applies. In other words, a publisher writing the value of a data object must have the means to uniquely indicate the data object it is writing. This way, the middleware can distinguish the instance being written and decide, for example, to keep only the most current value.

As shown in Figure 3, DDS uses the combination of a *topic* object introduced above and a *key* to uniquely identify data-object instances. The representation and format of the key depend on the data type. However, since a *topic* is bound to a unique type, the service can always interpret the key properly given the *topic* and the value of a data object.

The combination of a fixed-type *topic* and a *key* is sensible for data-centric systems because the *topic* represents a set of related data objects that are treated uniformly (e.g., track information of aircraft generated by a radar system) where each individual aircraft can be distinguished by the value of a data field (such as the flight number), which is interpreted as the *key*. Alternatively, a topic can be associated with a unique data stream (e.g., an Alert) in the case where the *topic* does not define any keys.

The presence and definition is made by the application and it can be different for each topic. It is possible for the key to be a single value within the data object (e.g., a serial number field) or a combination of fields (e.g., airline name and flight number). This use of a key is unique to data-centric systems and is used neither in "enterprise" publish/subscribe systems nor in event-distribution systems.

Quality of Service

A key aspect of the DDS standard is the pervasive use of Quality of Service (QoS) to configure the system and the introduction of middleware-brokered QoS contracts between publishers (who offer a maximum level for each QoS policy) and subscribers (who request a minimum level for each QoS policy).

QoS contracts provide the performance predictability and resource control required by real-time and embedded systems while preserving the modularity, scalability and robustness inherent to the anonymous publish/subscribe model. Table 1 lists some of the supported *QoS policies*.

In addition to the ones in Table 1, the following QoS policies are also supported: USER_DATA, TOPIC_DATA, GROUP_DATA, DURABILITY, PRESENTATION, LATENCY_BUDGET, OWNERSHIP, OWNERSHIP_STRENGTH, LIVELINESS, PARTITION, TRANSPORT_PRIORITY, LIFESPAN, DESTINATION_ORDER and RESOURCE_LIMITS.

Policies	
DEADLINE Parameters: A duration "deadline_period"	Indicates that a ***DataReader*** expects a new sample updating the value of each instance at least once every *deadline_period.* Indicates that a ***DataWriter*** commits to write a new value for each instance managed by the ***DataWriter*** at least once every *deadline_period.*
TIME_BASED_FILTER Parameters: A duration "minimum_separation"	Filter that allows a ***DataReader*** to specify that it is interested only in (potentially) a subset of the values of the data. The filter states that the ***DataReader*** does not want to receive more than one value each *minimum_separation*, regardless of how fast the changes occur.
CONTENT_BASED_FILTER A string "expression" and a sequence of strings "parameters"	Specified a filter that allows a ***DataReader*** to filter the data received from a given ***Topic*** based on the contents of the data itself. Syntax of "expression" is like an SQL WHERE clause. Only the parameter part may be changed.
HISTORY A "kind": KEEP_LAST, KEEP_ALL And an integer "depth"	Specifies the behavior of the Service in the case where the value of a data-object changes (one or more times) before it can be successfully communicated to one or more existing subscribers. The HISTORY policy controls whether the Service should attempt to keep in its history only the most recent set of values (KEEP_LAST), or all values (KEEP_ALL)
RELIABILITY A "kind": RELIABLE, BEST_EFFORT	Specifies whether the middleware should use a reliable protocol to ensure each data-sample present in the publisher history is received by all subscribers.

Table 1 Some of the QoS policies supported by DDS

Data, Message or Event Propagation

The information transferred by data-centric communications can be classified into Signals, States and Events/Messages.

Signals represent data that is continuously changing (such as the readings of a sensor). Signal publishers typically set the RELIABILITY QoS to best efforts and HISTORY QoS to KEEP_LAST.

States represent the state of a set of objects (or systems) codified as the most current value of a set of data attributes or data structures. The state of an object does not necessarily change with any fixed period. Fast changes may be followed by long intervals without change. Consumers of "state data" are typically interested in the most current state. Moreover, as the state may not change for a long time, the middleware will have to ensure that the most current state is delivered reliably. In other words, if a value is missed, then it is not generally acceptable to wait until the value changes again. State data publishers typically set the RELIABILITY QoS to reliable and HISTORY QoS to KEEP_LAST.

Events/Messages represent streams of the values that have individual meaning that is not subsumed by subsequent values. Events/Messages publishers typically set the RELIABILITY QoS to reliable and HISTORY QoS to KEEP_ALL

Many real-time applications have a requirement to model some of their communication patterns as a pure data-centric exchange where applications publish (supply or stream) "data" which is then available to the remote applications that are interested in it. These types of real-time applications can be found in C4I systems, industrial automation, distributed control and simulation, telecom equipment control and network management. Of primary concern to these real-time applications is the efficient distribution of data with minimal overhead and the ability to control QoS properties that affect the predictability, overhead and resources used. Distributed shared memory is a classic model that provides data-centric exchanges. However, this model is difficult to implement efficiently over the Internet. The OMG Data Distribution Service (DDS) has standardized a data-centric model to solve this situation. The specification also defines the entities, operations and QoS an application can use to create. ◢

Real-Time Innovations, Inc.
Sunnyvale, CA.
(408) 734-4200.
[www.rti.com].

Object Management Group.
[www.omg.org].