A Comparison and Mapping of Data Distribution Service (DDS) and Java Message Service (JMS)

Rajive Joshi, Ph.D.
Principal Engineer
Real-Time Innovations, Inc.
3975 Freedom Circle, Santa Clara, CA 94054
408-200-4754, rajive.joshi@rti.com

Abstract

Data-centric design is emerging as a key tenet for building advanced data-critical distributed embedded and enterprise systems. DDS and JMS are popular middleware API standards that are easy to use, and offer the benefits of using a publish-subscribe communication model resulting in loosely coupled scalable distributed applications. However, their differences have significant impact on a data-centric design.

DDS and JMS are based on fundamentally different paradigms with respect to data modeling, dataflow routing, discovery, and data typing; yet they offer a similar and easy to use experience to the application programmer. They differ significantly in their support for data filtering and transformation, connectivity monitoring, redundancy and replication, and delivery effort. Each also offers some distinct capabilities; and they both offer some equivalent capabilities. We provide a detailed functional comparison of the two standards, and discuss their implications on data-centric design.

We also discuss the practical considerations and differences in using the two standards. These include middleware architecture, platform support, interoperability, transports, security, administration, performance, scalability, real-time application specific support, and enterprise application specific support.

DDS and JMS APIs may be used together in an application. The can leverage each other via JMS-DDS bridging, JMS/DDS bindings, or by using DDS for JMS discovery. We discuss these approaches and their suitability for different data-centric integration scenarios.

DDS and JMS merit careful consideration for data-centric design and integration. Using one or both can considerably simplify data-centric development, and help maintain the focus on application issues, rather than becoming hijacked by communication and data delivery concerns.

Introduction

Emergence of data-centric design

Data-centric design is emerging as a key tenet for building advanced data-critical embedded and enterprise systems, as result of the growing popularity of cheap and widespread data collection "edge" devices, the easy availability of high performance messaging and database technology, and the increasing adoption of SOA and Web Services in the enterprise world. As computation and storage costs continue to drop faster than network costs, the trend is to move data and computation locally, using data distribution technology to move data between the nodes as and when needed.

Data-centric design is key to systems which exhibit some or all of the following five characteristics: (a) participants are distributed; (b) interactions between participants are data-centric and not object-centric; often these can be viewed as "dataflows" that may carry information about identifiable data-objects; (c) data is critical because of large volumes, or predictable delivery requirements, or the dynamic nature of the entities; (d) computation is time sensitive and may be critically dependent on the predictable delivery of data, (e) storage is local. Examples of data-centric systems are found in traffic control, command and control, networking equipment, industrial automation, robotics, simulation, medical, supply chain, and financial processing.

Several middleware technologies and standards have been applied to construction of distributed systems including DDS, JMS, EJB, HLA, CORBA, CORBA Notification Service. These middleware technologies fit the requirements of data-centric distributed systems to varying degrees. Specific requirements demanded by data-centric distributed systems include (1) ability to specify structured data models; (2) ability to dynamically specify and (re)configure the data flows; (3) ability to describe delivery requirements per data flow; (4) ability to specify and control middleware resources such as queues and buffering; (5) resiliency to individual node or participant failures; and (6) performance and scalability with respect to number of nodes, participants, and data flows.

Data-centric design with DDS and JMS

DDS and JMS are popular publish-subscribe middleware technologies that have been used to address the requirements of data-centric distributed-system design.

Data Distribution Service (DDS) is a formal standard from the Object management Group (OMG) popular in embedded systems, especially in industrial automation, aerospace, and defense applications. DDS specifies an API designed for enabling real-time data distribution. It uses a publish-subscribe communication model, and supports both messaging and data-object centric data models.

Java Message Service (JMS) is a defacto industry standard popular in the enterprise systems for messaging applications. JMS specifies a Java API for wrapping message-oriented middleware (MOM) APIs, so that portable application (Java) application code may be written. In that respect, it is similar to other Java APIs such as JDBC for abstracting database access, or JNDI for abstracting naming and directory services. JMS uses a publish-subscribe communication model, and a messaging or eventing data model.

DDS and JMS are similar in some respects. They both provide standardized APIs to preserve application portability across middleware vendors; both use a publish-subscribe (P-S) communication model. The P-S communication model (Figure 1), uses asynchronous message passing between concurrently operating subsystems. The publish-subscribe model connects anonymous information producers with information consumers. The overall distributed system is composed of processes, each running in a separate address space possibly on different computers. We will call each of these processes a “participant application”. A participant may be a producer or consumer of data, or both.

Publish-Subscribe Middleware

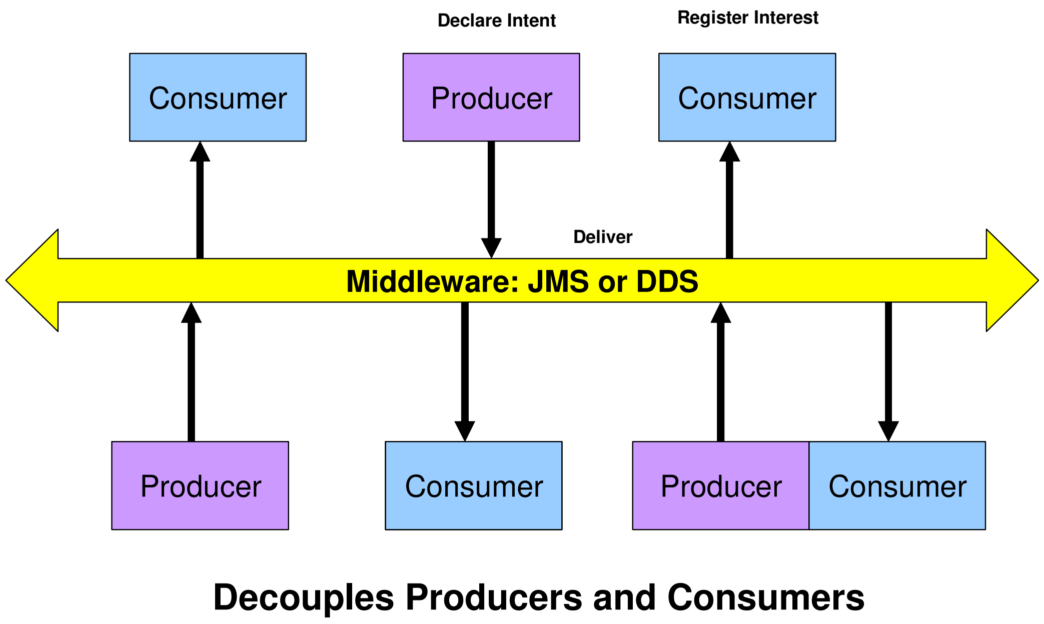

Figure 1 Publish-subscribe middleware decouples information producers from consumers.

Data producers declare the topics on which they intend to publish data; data consumers subscribe to the topics of interest. When a data producer publishes some data on a topic, all the consumers subscribing to that topic receive it. The data producers and consumers remain anonymous, resulting in a loose coupling of sub-systems, which is well suited for data-centric distributed applications.

Using DDS or JMS middleware can simplify distributed data-centric application design. The P-S communication model enables a robust service based application architecture that decouples participants from one another, provides location transparency, and flexibility to dynamically add or remove participants. Thus, DDS or JMS middleware often serves as the integration glue or the “data bus” interconnecting the participants producing or consuming data.

Both DDS and JMS APIs are intuitive and easy to use, and their popularity mitigates the risk in utilizing them for new data-centric designs.

DDS and JMS differ in their ability to cater to the key data-centric design requirements. We discuss these differences with respect to the requirements of data-centric systems including (1) data modeling and manipulation, including lifecycle management, data filtering, and transformation; (2) dataflow routing and discovery, including point to point connectivity; (3) delivery quality of service (QoS) per data flow, including delivery effort levels, timing control, ordering control, time-to-live, and message priority; (4) resource specification and management, including resource limits, and history; (5) resiliency to failures, including redundancy and failover, and status notifications; and (6) performance and scalability.

DDS is newer standard based on fundamentally different paradigms than JMS, with regards to data modeling, dataflow routing, discovery, and data typing; these differences enable applications designers with powerful new architectural possibilities. Despite these differences, the user experience of writing to DDS APIs is similar to that of JMS APIs. Also, they both provide support for persistent delivery, and time-to-live for a data item. .

DDS offers several enhanced capabilities with respect to data filtering and transformation, connectivity monitoring, redundancy and replication, and delivery effort. DDS offers new capabilities with respect to data-object lifecycle management, predictable delivery, delivery ordering, transport priority, resource management, and status notifications.

Distinctive DDS capabilities include data modeling and lifecycle management, automatic dataflow routing, spontaneous discovery, content based filtering and transformation, per dataflow connectivity monitoring, simple redundancy and replication, delivery ordering, and real-time specific features such as best efforts delivery, predictable delivery, resource management, and status notifications.

JMS offers some capabilities not offered by DDS. Distinctive JMS capabilities include point-to-point delivery to exactly one of many consumers, message priority, and enterprise specific features such as full transactional support, and application level acknowledgements.

DDS is amenable to a decentralized peer-to-peer architecture, which can be more robust and efficient compared to centralized server based architecture commonly used for JMS. Unlike JMS, which is a Java language standard, standard DDS APIs are available in many languages. Neither DDS nor JMS provide an interoperability protocol, although there is one currently under standardization for DDS. Neither specifies a transport model, although there are some capabilities in DDS that are better suited to unreliable transports such as UDP, while JMS can generally benefit from the availability of a reliable transport like TCP. Both DDS and JMS defer security to the application, and only provide support for communicating security credentials. Unlike DDS, JMS requires administration of the JMS provider (server) and JNDI registries. The API design choices made by DDS can support potentially higher performance (lower latency and higher throughput) and better scalability than JMS. DDS has some capabilities optimized for real-time applications, not found in JMS. JMS has some capabilities optimized for enterprise applications, not found in DDS.

DDS and JMS can be used simultaneously in an application. Infrastructure already invested in JMS can leverage DDS, and vice-versa. Possible approaches include: JMS-DDS bridging, JMS/DDS bindings, and using DDS for JMS discovery.

DDS and JMS merit careful consideration for data-centric design. Using one or both can considerably simplify a data-centric design, and help maintain the focus on application issues, rather than becoming bogged down by communication and data delivery concerns.

Background

We briefly summarize the key elements of the DDS and JMS middleware technologies.

DDS Synopsis

DDS targets real-time systems; the API and Quality of Service (QoS) are chosen to balance predictable behavior and implementation efficiency/performance. The DDS specification describes two levels of interfaces:

- A lower level Data-Centric Publish-Subscribe (DCPS) that is targeted towards the efficient delivery of the proper information to the proper recipients.
- An optional higher-level Data-Local Reconstruction Layer (DLRL), which allows for a simpler integration into the application layer.

The DCPS model builds on the idea of a "**global data space**" of *data-objects* that any entity can access. Applications that need data from this space declare that they want to subscribe to the data, and applications that want to modify data in the space declare that they want to publish the data. A data-object in the space is uniquely identified by its *keys* and *topic*, and each topic must have a specific type. There may be several topics of a given type. A global data space is identified by its *domain id*, each subscription/publication must belong to the same domain to communicate.

Figure 2 illustrates the overall data-centric publish-subscribe model, which consists of the following entities: ***DomainParticipant***, ***DataWriter***, ***DataReader***, ***Publisher***, ***Subscriber***, and ***Topic***. All these classes extend ***Entity***, representing their ability to be configured through QoS policies, be enabled, be notified of events via listener objects, and support conditions that can be waited upon by the application. Each specialization of the ***Entity*** base class has a corresponding specialized listener and a set of ***QoSPolicy*** values that are suitable to it.

Publisher represents the objects responsible for data issuance. A ***Publisher*** may publish data of different data types. A ***DataWriter*** is a typed facade to a publisher; participants use ***DataWriter(s)*** to communicate the value of and changes to data of a given type. Once new data values have been communicated to the publisher, it is the ***Publisher***'s responsibility to determine when it is appropriate to issue the corresponding message and to actually perform the issuance (the Publisher will do this according to its QoS, or the QoS attached to the corresponding ***DataWriter***, and/or its internal state).

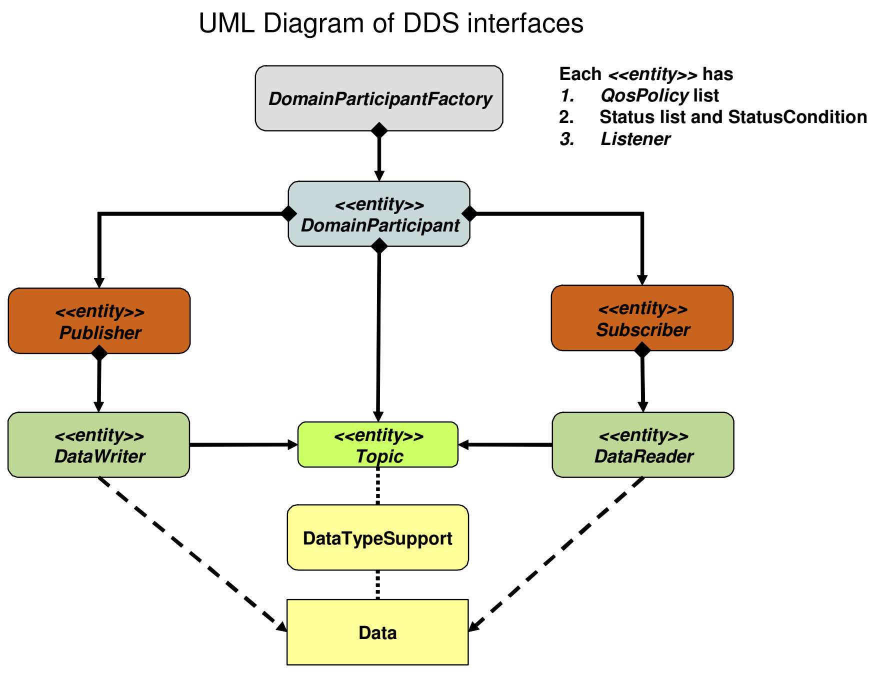

Figure 2 UML diagram of the DDS data-centric publish-subscribe interfaces

A ***Subscriber*** receives published data and makes it available to the participant. A ***Subscriber*** may receive and dispatch data of different specified types. To access the received data, the participant must use a typed ***DataReader*** attached to the subscriber.

The association of a ***DataWriter*** object (representing a publication) with ***DataReader*** objects (representing the subscriptions) is done by means of the ***Topic***. A ***Topic*** associates a name (unique in the system), a data type, and QoS related to the data itself. The type definition provides enough information for the service to manipulate the data (for example serialize it into a network-format for transmission). The definition can be done by means of a textual language (e.g. something like “float x; float y;”) or by means of an operational “plugin” that provides the necessary methods.

The DDS middleware handles the actual distribution of data on behalf of a user application. The distribution of the data is controlled by user settable Quality of Service (QoS).

JMS Synopsis

JMS targets enterprise messaging; the API is chosen to abstract the programming of a wide variety of message-oriented-middleware (MOM) products in a vendor neutral and portable manner, using the Java programming language.

Figure 3 illustrates the structure of the JMS API. A ***Destination*** refers to a *named* physical resource managed by the underlying MOM. It is administered and configured via vendor provided tools, and typically accessed by a user application via the Java Naming and Directory Interface (JNDI) APIs (external to JMS). A ***MessageProducer*** will send messages to a destination and a ***MessageConsumer*** can receive messages from a destination. The destination can be thought of a mini-message broker or a channel independent of the producers and consumers.

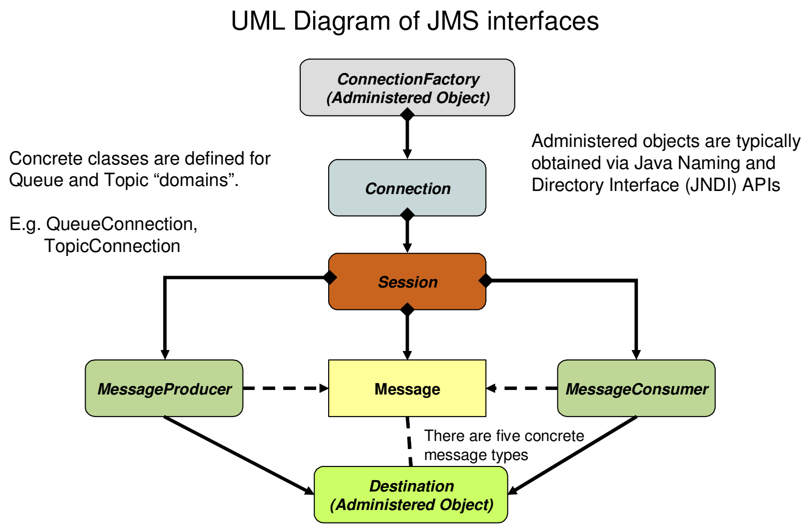
UML Diagram of JMS interfaces
ConnectionFactory (Administered Object)
Concrete classes are defined for Queue and Topic "domains".
E.g. QueueConnection, TopicConnection
Connection
Administered objects are typically obtained via Java Naming and Directory Interface (JNDI) APIs
Session
MessageProducer
Message
MessageConsumer
There are five concrete message types
Destination (Administered Object)

Figure 3 UML diagram of JMS messaging interfaces

JMS supports two different "messaging domains" (unrelated to the DDS domain concept) *point-to-point (PtP)* and *publish-subscribe (Pub/Sub).* The two messaging domains are provided to support the wide variety of MOM vendors; only one of them is required to be supported by a JMS provider, although many support both. They provide two different sets of derived classes that extend the common abstract APIs, as shown in Figure 4.

JMS Common	JMS PtP Domain	JMS Pub/Sub Domain
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber

Figure 4 The PtP and Pub/Sub JMS domains extend common abstract interfaces, and follow the same programming idioms.

The two JMS messaging domains are similar in every respect, except for the following ways.

1. In PtP messaging domain, only one consumer will receive a message; the policy is not specified by JMS and left up to the vendor. The messages are delivered in the order they are produced (as if put into a shared serial queue). Also, an application can peek ahead using a ***QueueBrowser.***
2. In the PtP messaging domain, the consumers are durable (see below), and therefore don't have to be running concurrently with the producers to receive messages. This can be achieved in the JMS Pub/Sub messaging domain by using durable subscriptions

A ***ConnectionFactory*** refers to vendor provided factory for ***Connection*** objects, and is also configured and administered using vendor provided tools, and typically obtained via JNDI APIs. An optional username, and password may be supplied when creating a ***Connection.***

A ***Connection*** is a heavy-weight object representing the link between the application and the middleware. Its attributes include a ***clientID***. It provides methods to ***start()*** and ***stop()*** communication and to ***close()*** a connection. An ***ExceptionListener*** may be registered with it, to trap lost connections. A ***Connection*** is used to create ***Session*** objects.

A ***Session*** represents a *single threaded context* for producing and/or consuming data*.* It provides methods to create ***Messages***, ***MessageProducers*** and ***MessageConsumers.*** Its attributes include whether it ***isTransacted*** and the ***acknowledgementMode***. In a transacted session, messages are not actually sent (MessageProducer) or the received messages not acknowledged (MessageConsumer) until a ***commit()*** operation. A ***rollback()*** operation can undo the pending messages to be sent (***MessageProducer***) or acknowledged (***MessageConsumer***). The ***acknowledgementMode*** determines whether received messages should be automatically acknowledged such that duplicates may (or may not) be received, or whether they must be explicitly acknowledged by the application by calling ***Message.acknowledge()***.

A ***Message*** is a first class object in JMS; it represents an event, and can carry an *optional* payload. A message is comprised of *headers*, optional user defined *properties*, and an optional user *data payload*. The JMS provider automatically assigns most message headers including: *destination, delivery mode, message id, timestamp, expiration, redelivery flag,* and *priority*. The user can assign some headers, including: reply to, correlation id, and type. In addition, the user can associate arbitrary properties consisting of (name, value) pairs. These properties can be used in '*selectors',* which are expressions specified on a ***MessageConsumer*** to sub-select and consume only the matching messages. JMS defines five message subclasses to conveniently specify the data payload. The message subclasses for unstructured payloads include ***TextMessage, ByteMessage,*** and ***ObjectMessage;*** and for structured payloads include ***StreamMessage*** and ***MapMessage***.

A ***MessageProducer*** is used to produce messages. A default destination may be specified when the producer is created; it can also be specified when sending messages. In addition, the delivery mode, priority, and expiration can be specified for the outgoing message headers. A persistent delivery mode means that a message will be delivered once-and-only-once; the message is stored in permanent storage before the ***send()*** method returns. A non-persistent delivery mode means that the message will be delivered at most once; a message may be dropped if the JMS provider fails.

A ***MessageConsumer*** is used to consume messages from a destination. A *selector* can be specified when creating a consumer; the consumer will only deliver the messages whose properties match the selector expression. Message can be delivered asynchronously by registering a ***MessageListener;*** the ***onMessage()*** method will be called when a message arrives. Alternatively, messages can also be received synchronously by calling ***receive*()*** methods, the desired timeout (zero, finite, infinite) can be chosen by the user. A consumer can be *durable*; for the Pub/Sub messaging domain this is specified by calling ***Session.createDurableSubscriber()*** and specifying a subscription name; in the PtP messaging domain, a ***QueueReceiver*** is always durable. A durable consumer receives all messages sent to a destination, including ones that are sent when the consumer is inactive. The JMS provider retains a record of the durable consumer(s) and ensures that all messages from the destination's producers are retained until the durable consumer acknowledges them or they have expired.

A ***Session*** can also create *unique* temporary destinations (***TemporaryQueue*** or a ***TemporaryTopic***), which are like administered destinations except that they are only valid for the duration of the connection and only the consumers associated with the connection can consume the messages***.*** However anyone can produce on the temporary destinations; their presence is typically conveyed to other producers using the ***Message.setReplyTo()*** method.

JMS-DDS Equivalents

We restrict our discussion of DDS to the DCPS layer, which has resemblances to JMS. There is no DLRL counterpart in JMS. A map of key JMS concepts and terminology and the DDS equivalents is summarized below. Additional details can be found in the following sections.

JMS	**DDS**
Client	Application
Provider = client runtime + server (if any)	Middleware, Service
Domains are PtP and Pub/Sub	Domain represents a global data space, comprised of a set of communicating user applications
ConnectionFactory	DomainParticipantFactory
Connection start()	DomainParticipant enable()
Session	Publisher, Subscriber

	FooTypeSupport extends TypeSupport Used to register a user type 'Foo' with a DomainParticipant
Destination A named physical resource that gathers and disseminates messages addressed to it	Topic (of type "Foo") An abstraction with a unique name, data-type, and QoS, used to connect matching DataWriters and DataReaders
Message	Foo (data-object) An instance of type 'Foo'
MessageProducer	FooDataWriter extends DataWriter
MessageConsumer	FooDataReader extends DataReader

Figure 5 Mapping of key JMS and DDS concepts and terminology.

Comparison of JMS and DDS

Fundamental paradigm differences

There are some fundamental conceptual differences between DDS and JMS, which deeply impact data-centric design. These differences are discussed below.

Data modeling: Autonomous messages vs. Data-objects

P-S middleware can be distinguished in their use of data models, which ranges from (1) messaging or eventing, where the data payload is opaque to the middleware; to (2) data-object centric, where the data payload is interpreted and managed by the middleware. Messaging or eventing P-S middleware treat a message on a topic as an event with an optional data payload that is opaque to the middleware. Data-object centric (or simply data-centric) P-S middleware allow an application to identify 'data-objects' to the middleware. The 'data-objects' are unique in the 'global data space' of the distributed system across all participants. Each participant is regarded as having a local cache of the underlying global data-object. A message on a topic is regarded as an update to the underlying data-object that can be identified and managed by the middleware. Local changes to a data-object are propagated by the middleware; the middleware can distinguish between messages or update samples from different data-objects and manage their delivery to the interested participants on a per data-object basis.

JMS does not support an underlying data model; it is a pure “messaging” or “eventing” middleware, and treats a message as an event with an optional data payload that is opaque to the middleware. A JMS message is a self-contained autonomous entity, representing an event with optional data payload. In its lifetime, a message may be (re)sent multiple times across multiple processes. A JMS client on the way may examine it, consume it, forward it, or generate new messages to accomplish its task. A message is uniquely identified with ***messageId,*** and carries with it its deliveryMode, priority, expiration, correlationID, redelivery flag, reply destination, and so on in the header fields. Message payload contents are not interpreted or managed by the JMS provider; each message is a unique and distinct entity. Data modeling capabilities, if needed, will have to be provided at the application layer, in the JMS client software.

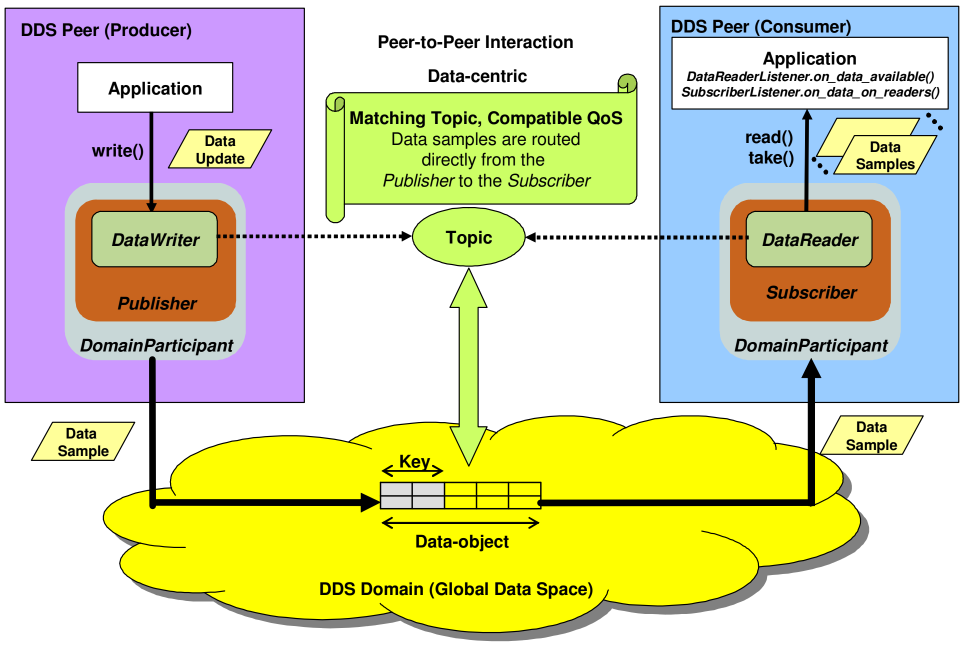

Figure 6 DDS provides a relational data model. The middleware keeps track of the data-objects instances, which can be thought of as rows in a table.

DDS is data-object centric middleware, and supports a ***relational data model*** commonly used in database applications, as illustrated in **Figure 6**. In database terms, a topic corresponds to a ***Table***; the table schema corresponds to the topic type. Certain type fields (columns) can be marked as ***keys*** (primary keys) in the type description (table schema). A data-object instance is identified by its keys, and corresponds to a row in the table. Underlying this data model is an implicit assumption of a shared *global data space* in which the data-objects live. The global data space is defined by the communicating applications in the DDS domain. Each participant is viewed as having access to a local cache of the topics (tables) in the global data space. A DataWriter can write (or update) one or more data-object instances (or rows) in its local cache. The updates are propagated by the middleware to the associated DataReaders for the topic, and are delivered as samples to be applied to the local cache on the receiving end. The DDS middleware can distinguish between different data-object instances based on the keys, and can manage the delivery of samples on a per data-object instance basis. Since the keys are embedded in the data type, relations between data-object instances are also implicitly managed by the DDS middleware.

DDS also supports unkeyed topic types, which are effectively equivalent to messaging (or eventing), as supported by JMS.

Unlike JMS, where messages are first class objects, DDS messages are user defined types and do not carry any 'per message' user settable headers or fields. However, the user is free to define the message data type, and therefore can specify needed fields.

As a consequence of this difference, DDS data delivery has the potential to be higher performance than JMS messages delivery, because the extra overhead of manadatory headers per message is not required with DDS.

Dataflow routing: Specific destinations vs. Matching endpoints

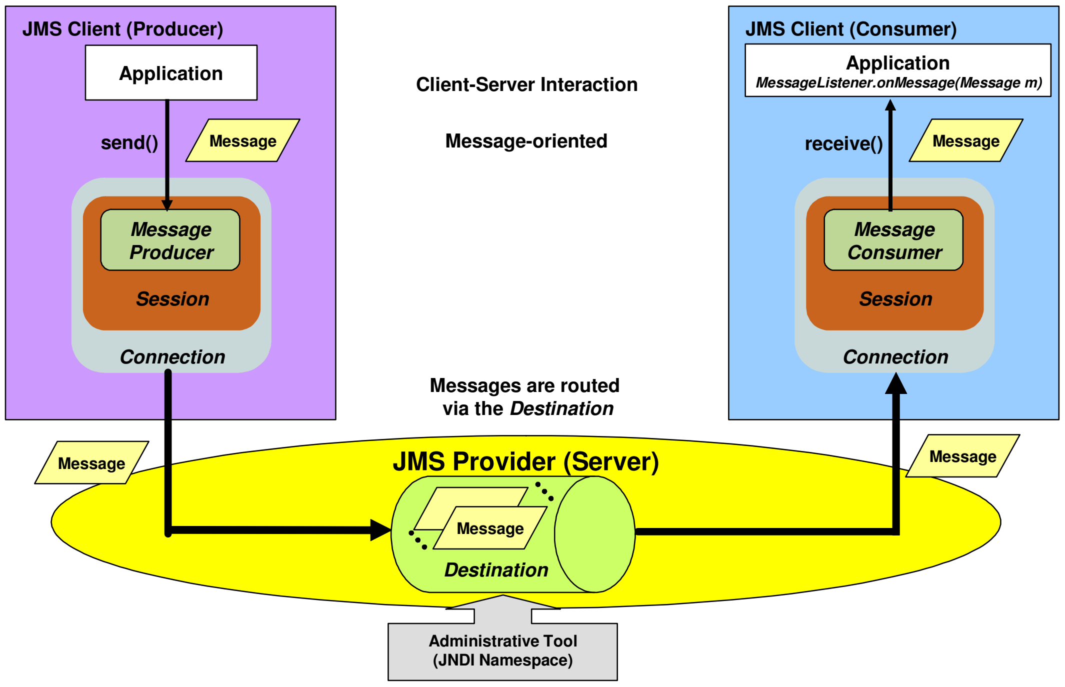

Figure 7 JMS destinations are logical message stores or channels configured using administrative tools supplied by the JMS vendor.

JMS destinations (Queue or Topic) are logical "message stores or channels", uniquely defined and managed by the middleware, as shown in **Figure 7**. A destination may be configured statically in the middleware using JMS vendor provided configuration tools; or it may be created dynamically using temporary destinations. In either case, they represent unique well-defined "channels" in the middleware. A destination and can hold any type of message (since JMS is opaque to the payload). A consumer is attached to a specific destination from which it will receive messages. A producer can specify the destination at the time of sending a message. A destination acts as a "mini-broker" managing the delivery of the messages sent to it. A dataflow is established between a producer and a consumer via the destination as the intermediary.

A DDS topic represents an association between compatible DataWriters or DataReaders bound to the topic, in the global data space. A topic has a name, type, and associated QoS. An *endpoint* (DataReader or DataWriter) is tightly bound to a specific topic and may additionally specify different desired QoS. A dataflow between a DataReader and DataWriter is only established when the type and QoS offered by the DataWriter is compatible with that requested by the DataReader (Figure 8).

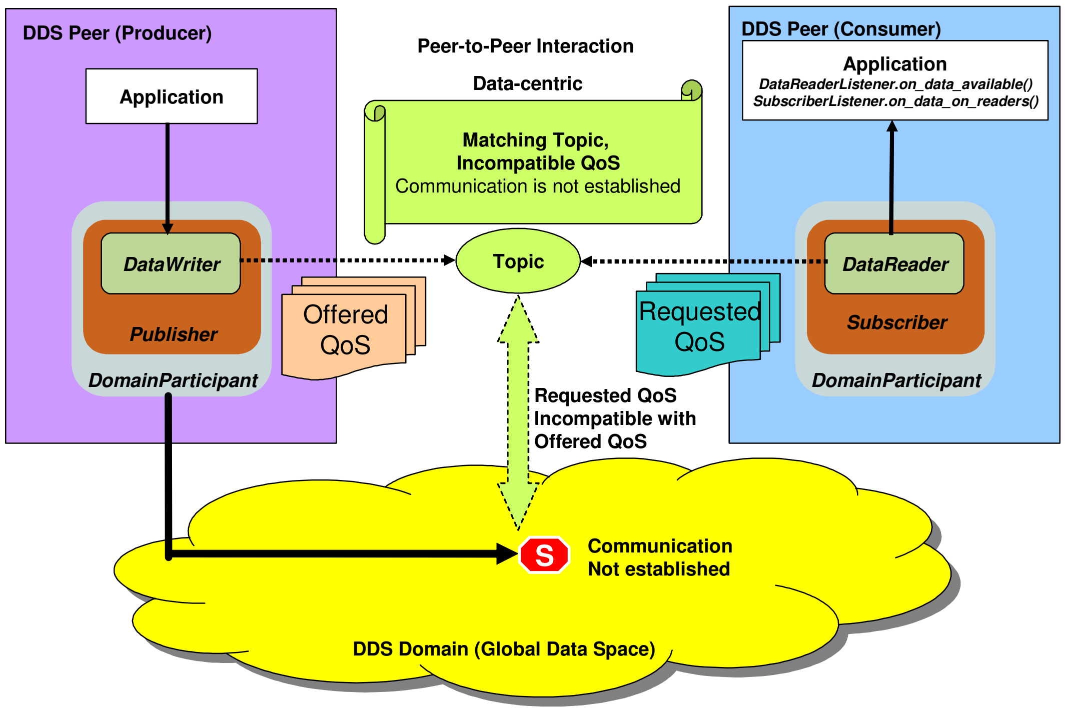
DDS Peer (Producer)
Application
DataWriter
Publisher
DomainParticipant
Offered QoS
Peer-to-Peer Interaction
Data-centric
Matching Topic, Incompatible QoS
Communication is not established
Topic
Requested QoS Incompatible with Offered QoS
DDS Peer (Consumer)
Application
DataReaderListener.on_data_available()
SubscriberListener.on_data_on_readers()
DataReader
Subscriber
DomainParticipant
Requested QoS
S
Communication Not established
DDS Domain (Global Data Space)

Figure 8 DDS topics represent a name, type, and QoS. DDS provides a spontaneous connection mechanism, which automatically connects matching DataReaders and DataWriters.

The DDS *requested/offered* mechanism establishes dataflows only between matching endpoints associated with a topic in the global data space. DDS notifies the application of incompatible endpoints, when a dataflow cannot be automatically established. Thus, DDS middleware truly acts like an “information bus”, where dataflows are dynamically established and removed.

Unlike JMS, where a producer sends to a *specific destination*, a DDS DataWriter (producer) never specifies a destination; in DDS the dataflows are automatically managed by the DDS middleware based on matching subscriptions. A DDS middleware implementation can take advantage of this behavior by supporting direct data transfer from a DataWriter to a DataReader, without involving an intermediary; thus it has the potential for better performance and scalability than JMS.

Discovery: Administered vs. Spontaneous

JMS discovery is *administered* and centralized. JMS discovery requires that the producers and consumers be able to find and bind to the destinations (and not each other). There are two mechanisms for JMS destination discovery.

- Static destinations are discovered via JNDI APIs, which bind logical destination names to destination objects. The static destinations accessible this way must have been previously configured in the JMS middleware (server) using vendor supplied administrative tool (**Figure 7**).
- Destinations (including temporary destinations) may also be discovered via the ***replyTo*** attribute of received messages. In order to discover a destination using this mechanism, a static destination must have already be previously established.

Since JMS discovery is administered, the static destinations must be determined and configured before a client can use them. Determining what static destinations to use is a critical aspect of a distributed system design, and must be considered carefully prior to deploying a system based on JMS. Evolving the system configuration for new requirements also requires careful planning and administration. Destinations take up physical resources, so destinations no longer needed in distributed system must be purged, and new ones added as needed over the lifetime of a distributed system based on JMS.

DDS discovery is *spontaneous* and decentralized. DDS requires that endpoints be able to find each other to determine if they are compatible and whether a dataflow should be established (Figure 8). Thus, discovery is implicit in the dataflow routing mechanism.

DDS provides APIs for an application to access the internal middleware discovery meta-data by means of **built-in topics.** The internal meta-data that can be accessed by a user application includes information such as participants joining/leaving a domain, creation/deletion of topics, data readers, and data writers in a domain. The DDS ***DomainParticipant.get_builtin_subscriber()*** method can be used to monitor the following builtin-topics: DCPSParticipant, DCPSTopic, DCPSPublication, DCPSSubscription.

Since DDS discovery is spontaneous, the topics can dynamically change over the lifetime of a deployed distributed system based on DDS, without any administrative impact. Endpoints on new topics are discovered automatically, and dynamic dataflows established in a *plug-n-play* fashion. The spontaneous discovery mechanism of DDS can also potentially scale better as the span of a distributed system grows.

Data typing: Predefined message types vs. Arbitrary user data types

JMS provides five predefined message types, to conveniently specify different types of message payloads. Since JMS destinations are not typed, any type of payload can be produced and consumed on a destination. If a consumer has a different idea of the message payload than the producer, it will manifest as runtime typecasting exception when the consumer tries to access the payload using a different message type. Also, the user data payload must be converted into one of the available message types, thereby involving conversion overhead between user data type and JMS message types at both the producer and consumer ends.

DDS does not provide any predefined message or data types. Instead it uses the data types defined in the programming language. Typically these are specified using interface definition language (IDL) in a programming language neutral way. Middleware vendor provided tools are used to generate a programming language type, and corresponding type support classes. For example, given a user type ***Foo,*** type specific ***FooTypeSupport, FooDataWriter,*** and ***FooDataReader*** are generated with APIs as per the DDS standard. This approach has several advantages: it allows for higher performance by eliminating a potential extra conversion between a user type and a middleware type; it potentially enables the user to plugin their own data serialization ad deserialization scheme. Also, since DDS topics are strongly typed, the middleware can detect a type mismatch between the endpoints and notify the application.

User experience similarities

Despite the fundamental paradigm differences, the DDS and JMS user experience is somewhat similar, making it relatively easy to understand and switch back-and-forth between the two programming models. Figure 9 illustrates the key steps in writing a JMS client (application). Figure 10 illustrates the key steps in writing a DDS application. The steps needed to write a user application are summarized below.

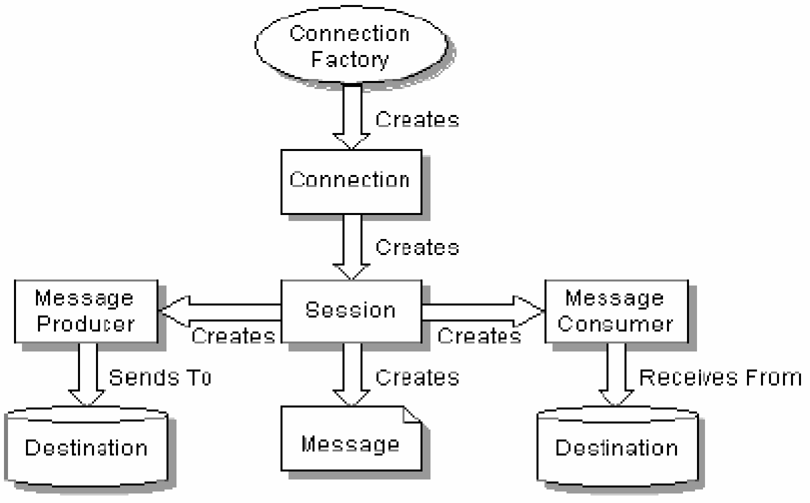

Figure 9 JMS programming model.

DDS Programming Model

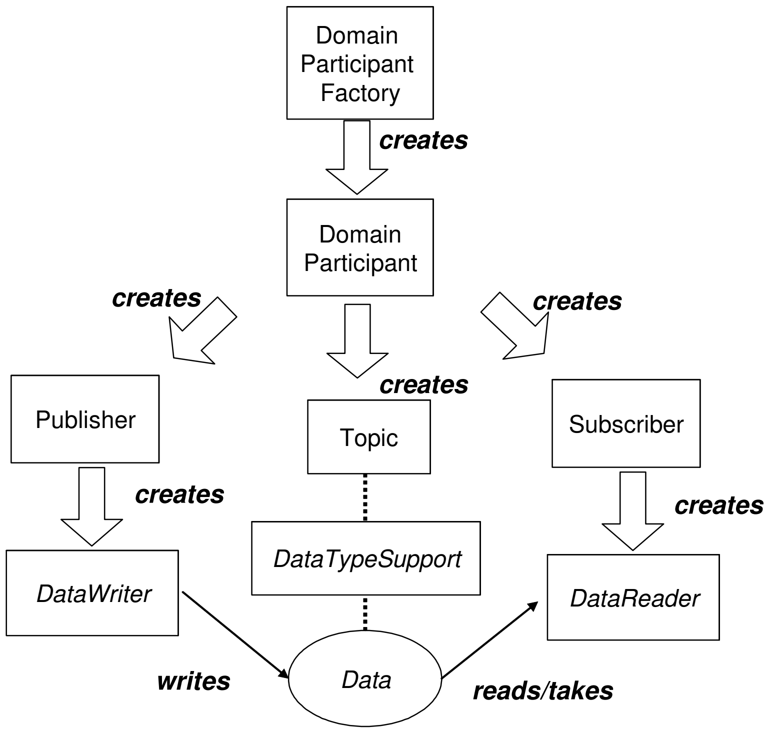

Figure 10 DDS programming model.

Step	**JMS (**Figure 9**)**	**DDS (**Figure 10**)**
0	Decide messaging domain: PtP or Pub/Sub.	Decide *domain id*, which represents a global data space, and isolates communication relative to other domains.
1	Get the ***ConnectionFactory*** from the Environment, typically using JNDI.	Get the ***DomainParticipantFactory,*** which is a singleton class.
2	Create a ***Connection.*** Set the ***clientID*** if needed. May also specify an ***ExceptionListener*** if needed.	Create a ***DomainParticipant***, specifying the QoS associated with it, and optionally a listener.

3	Create a ***Session***. Decide if the session should be transacted, and the acknowledgement mode to use.	Create a ***Publisher*** (for a producer application) or a ***Subscriber*** (for a consumer application). Specify the QoS and optionally a listener.
4a		Register the user type with the domain participant. A user type ***Foo*** is registered under the name "Foo" by calling ***FooTypeSupport.register_type(participant, "Foo")***
4b	Get a ***Destination.*** An administered destination is typically obtained using JNDI. Alternatively, a temporary destination can be created from the ***Session*** object.	Create a named ***Topic*** from the domain participant for the registered user type. Specify the QoS associated with the topic, and optionally a listener. Use a ***ContentFilteredTopic*** to deliver a sub-set of samples that meet a certain selection criteria; and a ***MultiTopic*** to combine and transform received samples on various topics into a desired format.
5	***Producer***	***Producer***
5a	Create ***MessageProducer***, specifying the default ***Destination*** on which it will send messages.	Given a topic of type ***Foo,*** create a ***FooDataWriter*** bound to it. Specify the QoS and optionally a listener.
5b	Create a ***Message*** of the sub-type appropriate for the data payload. Optionally, set the ***replyTo*** attribute to specify the ***Destination*** on which the client wants a reply. Also specify custom message properties as appropriate.	
5c	Set the message data payload (if any)	
5d	Send the message. Specify the***, deliveryMode, priority, and expiration.*** Optionally specify an alternative ***Destination.***	Write an instance of user type ***Foo*** using the ***FooDataWriter.***

6	***Consumer***	***Consumer***
6a	Given a ***Destination,*** create a ***MessageConsumer*** for it. Optionally specify *selectors* to sub-select messages, and decide whether local messages from producers on this connection should be ignored or not. If using a Pub/Sub messaging domain, also decide whether the subscription should be durable (survive consumer failures and inactivity periods).	Given a topic of type ***Foo,*** create a ***FooDataReader*** bound to it. Specify the QoS and optionally a listener.
6b	Decide whether to consumer messages asynchronously or synchronously. For asynchronous delivery, register a ***MessageListener*** and process the incoming messages in its ***onMessage()*** method. For synchronous delivery, call the appropriate ***receive()*** method with desired timeout (if any).	Decide whether to receive updates asynchronously or synchronously. For asynchronous delivery, register a ***DataWriterListener*** and process the incoming updates in its ***on_data_available()*** method. For synchronous delivery, use a ***WaitSet*** to wait for data to become available, specifying the desired timeout (if any).
6c	Process the message. May want to look at the ***replyTo*** attribute find out the destination on which the producer is expecting a reply. If a reply is sent (Step 5), may set the ***correlationID*** attribute of the outgoing message.	Process the received samples. Since the type ***Foo*** is user-defined, we can specify fields in the user type to convey a reply topic, or a correlation id, or other attributes as needed.
6d	If using a PtP messaging domain, may use a ***QueueBrowser*** to peek ahead at the messages in the ***Queue.***	Can use ***FooDataReader*** to peek ahead using the ***read()*** methods. Samples are removed (or taken) from the middleware using the ***take()*** methods.

Capability differences

DDS and JMS offer distinct capabilities that impact deeply impact the architectural design of applications utilizing them.

Data filtering and transformation

In JMS, arbitrary (name, value) property pairs can be tagged on a message. The properties can be used in *selector* expressions specified on a consumer. This allows a consumer to sub-select the messages that are delivered to the application. Note that the message filtering mechanism does not look at the data contents; instead it is solely based on the properties of the message. JMS does not provide data transformation capability.

DDS provides two constructs (1) ***ContentFilteredTopic*** for filtering data samples on a topic based on its contents; and (2) ***MultiTopic*** for transforming data samples received on multiple topics into a new data representation. These constructs simplify data management, since they operate directly on the user data model, and unlike JMS, not on separately maintained properties associated with the data.

As a consequence, DDS middleware has the potential to be higher performance than JMS, since there is no extra overhead of computing, setting, and communicating additional message properties.

Connectivity monitoring

Connectivity monitoring refers to ability of an application to determine that it is no longer communicating with other endpoints or participants.

JMS supports ***ExceptionListener*** on a ***Connection*** to notify an application that it has lost the connection with the middleware.

DDS supports several QoS levels for detecting loss of connectivity, via the *liveliness* mechanism. A DataWriter is considered to be alive and connected to a DataReader if it asserts its liveliness within a *lease_duration.* The liveliness be asserted automatically by the middleware; as a side-effect of some user operation on the DomainParticipant; or explicitly by calling ***assert_liveliness().***

The LIVELINESS QosPolicy specifies the lease duration, and the method of asserting the liveliness.

When a DataWriter does not assert its liveliness within its lease period, the application is notified via the ***DataReaderListener.on_liveliness_changed()*** method. The application can install a listener on the “DCPSParticipant” builtin topic to detect if there are any participants communicating with it.

While JMS supports a simple mechanism for detecting connectivity loss with the JMS provider (and therefore other participants); DDS supports detection of connectivity loss between producer and consumer endpoints, as well as participants. This simplifies the design of fault-tolerant applications, and enables DDS applications to have *self-healing* qualities.

Redundancy and replication

In JMS, all the messages produced on a destination are seen by a consumer. Setting up redundant and replicated producers (primary and secondary) requires application level coordination and synchronization between the primary and secondary producers.

DDS provides two QoS policies that, combined with the spontaneous discovery, make it very simple to setup redundant replicated data producers. The OWNERSHIP QoSPolicy determines whether a DataReader will receive updates of data from just one DataWriter (the strongest), or from any associated DataWriter. The OWNERSHIP_STRENGTH QoSPolicy determines if a DataReader is to receive from only the strongest DataWriter; this QoSPolicy is set on each DataWriter and the one with the largest strength number will be the DataWriter a matching DataReader receives from.

Thus, a primary DataWriter and a replicated secondary DataWriter can be setup with ownership set to exclusive (i.e. updates will be received from only one DataWriter). The primary DataWriter is assigned higher strength than the replicated secondary writer. When the primary fails (connectivity is lost), the secondary writer will seamlessly take over. When the primary is restarted, it will shadow the secondary producer as long as it is active. This simple redundancy and replication mechanism can scale well for a large system, and gives a *self-healing* quality to DDS applications.

Delivery effort

JMS always attempts to deliver a message and receive an acknowledgement. JMS will attempt to redeliver a message (marking it as such) until the receiving end acknowledges it.

Like JMS, DDS also supports acknowledged delivery mode. In addition, it also supports a “best efforts” mode, in which messages are not acknowledged by the receiving end. This behavior may be specified via a RELIABILITY QosPolicy on a DataWriter, DataReader, or a Topic. This policy determines whether a message should be sent best effort (send once without expecting acknowledgements) or reliably (resent until positively acknowledged).

The DDS best efforts mode enables data to be transferred with minimal latency, and is well suited to the needs of many high-performance real-time sensor based applications.

User meta-data

User meta-data refers to the ability to associate additional user-specified information with the data delivered by the middleware.

JMS supports user meta-data by means of message properties, which are (name, value) pairs that can be specified on a per message basis.

DDS supports user meta-data by providing a USER_DATA QosPolicy, which allows an application to associate arbitrary information with the DataReader, DataWriter endpoints, or the DomainParticipant. The user meta-data is accessible via the built-in topics. The contents of the USER_DATA QosPolicy are mutable, and can be changed as needed.

The DDS approach for associating user meta-data with the endpoints, rather than per message can potentially support higher performance, as the user meta-data does need to be set and transferred on a per message basis.

Note that in DDS, per message properties can be achieved, by creating a wrapper user data type that contains a sequence of (name, value) pairs. This can be done in the application code.

Delivery acknowledgements

JMS messages always require acknowledgement. The acknowledgement mode is specified on a per session basis. It may be automatic (AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE) or explicit (CLIENT_ACKNOWLEDGE). In the CLIENT_ACKNOWLEDGE mode, a message must be explicitly acknowledge by the application by calling ***Message.acknowledge().*** The CLIENT_ACKNOWLEDGE mode is useful for guaranteeing end-to-end message delivery.

DDS sample updates are only acknowledged when the RELIABILITY QosPolicy is set to RELIABLE message delivery. The acknowledgements in DDS are automatic; there is no explicit method for a user application to acknowledge a received sample, and thereby indicate that it has actually consumed it.

Transactional behavior

A transaction allows a group of operations to be treated as a single unit of work. Either none or all the operations in the group are executed as a unit. If a transaction is rolled back, or one of the operations in the group fails, none of the operations take effect. Otherwise, all the operations take effect when a transaction is committed.

JMS supports transacted sessions. A MessageProducer in a transacted session actually sends the messages when a ***Session.commit()*** is called. Likewise, a MessageConsumer acknowledges the received messages only when the session is committed. A new transaction implicitly begins after the last commit. The ***Session.rollback()*** method may be called to undo the uncommitted messages waiting to be sent by a MessageProducer or acknowledged by a MessageConsumer. Send operations may be mixed with receive operations in a transaction.

The scope of a JMS local transaction is limited to the session. JMS supports the Java Transaction API (JTA) so that a JMS connection or a session can be used with a JTA compliant transaction manager to participate in a distributed transaction, using a *two or three-phase commit* protocol.

DDS partially supports transactional behavior for sending data via the notion of a set of “coherent changes”. A *coherent set* of changes is a set of modifications that must be propagated in such a way that they are interpreted at the receiver's side as a consistent set of modifications; that is, the receiver will only be able to access the data after all the modifications in the set are available at the receiver end. A coherent set of changes behaves as if sent atomically; if an event occurs that prevents a subscriber from receiving the entire set of coherent changes, it must behave as if it had received none of the set.

A coherent set of changes bracketed by calls to the ***Publisher.begin_coherent_changes()*** and ***Publisher.begin_coherent_changes()*** methods. The PRESENTATION QosPolicy controls the scope within which changes to set of data-object instances are considered coherent, and whether the ordering within that scope should be preserved.

The support for coherent changes enables a publishing application to change the value of several data-instances that could belong to the same or different topics and have those changes be seen atomically by the readers. This is useful in cases where the values are inter-related (for example, if there are two data-instances representing the altitude and velocity vector of the same aircraft and both are changed, it may be useful to communicate those values in a way the reader can see both together; otherwise, it may e.g., erroneously interpret that the aircraft is on a collision course).

DDS's support for transactional behavior is partial. It does not provide a way for an application to rollback a coherent set of changes. Also, DDS does not provide support facilities for participating in distributed transactions, using JTA compliant or other transaction processing monitors.

Point-to-point delivery

In the JMS PtP messaging domain, a destination (Queue) may have multiple consumers (QueueReceivers) and producers (QueueSenders). A message is processed by exactly one of the attached consumers. Thus, a message is delivered *point-to-point*, from the producer to one of the many available consumers. The policy for selecting a consumer is left up-to the middleware provider. Upon message redelivery (if any), a message may get dispatched to a different consumer. The point-to-point delivery mechanism in the JMS PtP messaging domain makes it very easy to distribute processing load across multiple identical consumers, thus providing a simple means for *load balancing*. However, since PtP behaves as if the messages are put in a single logical queue and handed over to one of the available consumers, this messaging domain will generally be less scalable that the Pub/Sub domain.

DDS does not support a point-to-point delivery mechanism. All the matching consumers associated with a topic will receive updates. The PARTITION QosPolicy may be used to *partially* achieve point-to-point delivery. A PARTITION QosPolicy, specifies a set strings that introduce a logical partition among the topics visible by a Publisher and a Subscriber. A DataWriter within a Publisher only communicates with a DataReader in a Subscriber if (in addition to matching the Topic and having compatible QoS) the Publisher and Subscriber have a common partition name string. A change of this policy can potentially modify the "association" of existing DataReader and DataWriter entities. It may establish new "associations" that did not exist before, or break existing associations. Point-to-point delivery may be accomplished by: (1) assigning a unique partition name to every consumer (Subscriber, DataReader pair); and (2) switching a producer (Publisher, DataWriter pair) among those partition names. The consumer selection policy can be configured in a variety of ways, at the application level.

Delivery priority

JMS messages have a priority header. The value of this header can be set directly on a message, or specified as a property of the producer. Higher priority messages are delivered ahead of lower priority messages.

DDS does not support the notion of delivery priority on data updates.

The priority is a hint in JMS. The middleware is not required to deliver the messages in priority order.

Capability equivalents

For certain capabilities, DDS and JMS offer equivalent ways of achieving the same results.

Persistency and Durability

Persistency refers to the ability of specifying data delivery so that it survives middleware failures. With persistent delivery, an application is assured that when a send or a write operation returns, the middleware will not lose the data even if it crashes.

Durability refers to the ability of a consumer to receive data sent to its destination or topic even when it is not active. After a durable consumer starts, either for the first time or after a crash, it receives any messages destined for it (while it was inactive).

JMS provides independent mechanisms for controlling both persistency and durability. In JMS durability is specified by creating a durable consumer: either by creating a ***QueueReceiver***, or by calling ***Session.createDurableSubscriber().*** Persistency is specified by ***deliveryMode*** used to send message. For PERSISTENT delivery mode, a message is persisted on permanent storage before the send method returns to the caller; the message is delivered *once-and-only-once* (redelivered messages are marked). For NON_PERSISTENT delivery mode, a message is not persisted before the send operation returns to the caller; the message is delivered *at-most-once* (allowing for the possibility of loosing a message after the send operation has completed, but before it can be delivered because of middleware failure). Note that NON_PERSISTENT messages may be saved on permanent storage anyway, in order to support durable consumers.

DDS also provides a means for independently controlling persistency and durability. The DURABILITY QosPolicy determines whether or not the middleware should save already-sent samples in case new a DataReader joins the later. There are several kinds of durability settings: VOLATILE to indicate that a DataReader will not receive any samples missed while it was inactive; TRANSIENT_LOCAL to indicate that a late joining DataReader will receive missed samples only from the DataWriters that are still active; TRANSIENT to indicate that a late joining DataReader will receive missed samples as long as the middleware has not crashed; and PERSISTENT to indicate that a late joining DataReader will receive missed samples even if the middleware crashed.

The DDS DURABILITY QosPolicy can be independently specified on DataWriters, DataReaders, and Topics. Persistency is assured by the PERSISTENT setting of the QosPolicy on a DataWriter. Durability (with different levels of service) is assured by TRANSIENT_LOCAL, TRANSIENT, or PERSISTENT settings of the QosPolicy on a DataReader.

Compared to JMS, DDS provides several levels of quality of service to controlling persistency and durability.

Time to live

The “time to live” for a message or data sample specifies how long it is valid.

JMS messages have an ***expiration*** header. The value of this header can be set directly on a message, and specified as a property of the producer.

DDS provides a LIFESPAN QosPolicy on a DataWriter and Topic, which specifies how long the data written by a DataWriter is considered valid.

In either case, messages or data samples that are no longer valid are automatically purged by the middleware.

New capabilities in DDS

DDS is a newer standard that addresses a broad range of data-centric design requirements; it has had the benefit of JMS hindsight. DDS supports some capabilities that have no counterpart in JMS.

Data-object lifecycle management

DDS middleware is cognizant of the underlying relational data model, and provides facilities that enable it to manage the lifecycle of data-object instances. An application can express the intent to produce updates to a data-object instance by calling ***DataWriter.register_instance()***; and conversely negate this intent by calling ***DataWriter.unregister_instance().*** A ***DataWriter.dispose()*** can be used to indicate that a data-object instance is deleted; it is analogous to deleting a row in a table.

A DataReader keeps track of a data-object instance’s status which can be ALIVE meaning there are connected DataWriters that may update it, NOT_ALIVE_NO_WRITERS meaning that there are no connected DataWriters that may update it, and NOT_ALIVE_DISPOSED meaning that that it has been explicitly disposed by a DataWriter.

The READER_DATA_LIFECYCLE QosPolicy specifies how long the DataReader must retain information regarding data-object instances that have the state NOT_ALIVE_NO_WRITERS.

The WRITER_DATA_LIFECYCLE QosPolicy controls whether or not a DataWriter will automatically dispose instances each time they are unregistered.

Predictable delivery

DDS provides QosPolicies specifically targeted to minimum latency, predictable real-time operation in high-performance distributed data-critical systems.

The DEADLINE QoSPolicy expresses the maximum duration (deadline) within which a DataReader expects a data-object instance to be updated. If a sample is not received within the deadline, a listener method is called.

The TIME_BASED_FILTER QosPolicy specifies a *minimum_separation* value that allows a DataReader to specify that it interested only in (potentially) a sub-sampled set of the values for a data-object instance. A DataReader does not want to receive more than one sample each *minimum_separation* for a data-object instance, regardless of how fast the changes occur at a DataWriter.

The LATENCY_BUDGET QosPolicy provides a hint as to the maximum acceptable delay from the time the data is written to the time it is received by the subscribing applications.

Delivery ordering

DDS provides QosPolicies to control the ordering of received samples.

The DESTINATION_ORDER QosPolicy controls the criteria used to determine the logical order among changes made by different Publishers to the same data-object instance. The order can be by reception timestamp or by source timestamp when a sample was written.

The PRESENTATION QosPolicy specifies how a coherent set of samples representing changes to data-object instances made by a single Publisher are presented to a subscribing application. This policy affects the application's ability to: specify and receive coherent changes, see the relative order of coherent changes.

Transport priority

DDS provides a TRANSPORT_PRIORITY QoSPolicy in a DataWriter, which allows a DDS application to take advantage of transports that are capable of sending messages with different priorities.

The priority is a hint in DDS, and may be used for traffic shaping. The behavior is transport and middleware dependent, and the middleware is not required to deliver higher transport priority updates first.

Resource management

DDS provides QosPolicies to specify the memory resources used by the middleware for sending and receiving samples.

The RESOURCE_LIMITS QosPolicy specifies the resources that the middleware can utilize in order to meet the requested QoS. The middleware will perform the data delivery within the confines of the resource limits.

The HISTORY QosPolicy specifies the behavior of the middleware in the case where the value of a sample changes (one or more times) before it can be successfully communicated to one or more existing consumers. On the publishing side this policy controls the number of samples per data-object instance that should be maintained by a DataWriter on behalf of the associated DataReaders. On the subscribing side it controls the number of samples per data-object instance, that should be maintained until the application "takes" them from the middleware by calling a ***DataReader.take()*** method.

The settings of the HISTORY QosPolicy must be within the confines of the RESOURCE_LIMITS QosPolicy.

Status notifications

DDS specifies a number of *status changes* that can trigger a listener invocation on a DDS ***Entity***. The list of status change notifications includes: INCONSISTENT_TOPIC, OFFERED_DEADLINE_MISSED, REQUESTED_DEADLINE_MISSED, OFFERED_INCOMPATIBLE_QOS, REQUESTED_INCOMPATIBLE_QOS, SAMPLE_LOST, SAMPLE_REJECTED, DATA_ON_READERS, DATA_AVAILABLE, LIVELINESS_LOST, LIVELINESS_CHANGED, PUBLICATION_MATCHED, SUBSCRIPTION_MATCHED. The availability of data is only one of the possible status notifications. Also, note that data samples are not delivered in a notification. A listener must call ***read()*** or ***take()***on a ***DataReader*** to access the data samples. This architecture make it possible to implement a potentially lower end-to-end latency middleware, since the extra overhead of determining which samples are to be delivered is not needed before invoking a listener.

Practical considerations

We have examined the functional similarities and differences between DDS and JMS, and their implications for distributed application design. In addition, a number of practical factors come into play when choosing a middleware technology for building a distributed system. These are discussed next.

Architecture

JMS APIs are described in terms of a client/provider interaction; the client being the user application, distinct from the JMS provider or the middleware server. Most popular JMS provider implementations have centralized server-based architecture; some use a cluster of servers for fault-tolerance and load balancing. In a centralized server-based architecture, a message must pass via the server, which introduces extra latency and a potential resource bottleneck.

DDS APIs are described in terms of a peer-to-peer interaction; data is transferred directly from a DataWriter to a DataReader. DDS implementations generally use a decentralized peer-to-peer architecture. For example, "RTI Data Distribution Service" from Real-Time Innovations, Inc (RTI) has a completely symmetric architecture. There is no single point of failure and data transfer latency is minimized. Participants can freely join or leave a domain, without needing special configuration. This is in keeping with the goals of DDS to enable robust, high-performance, low latency distributed applications.

Platforms

JMS, as the name implies, was developed to provide a portable vendor neutral *Java* API for a wide range of MOM implementations. JMS requires the Java platform. Some vendors provide JMS like APIs for other programming languages, but there is no established standard. JMS vendors may also provide proprietary APIs in other languages, native to the underlying MOM implementation. Also, note that JMS applications require non JMS APIs to bootstrap the client application. The standard practice is to use JNDI APIs, which are well established for Java EE programming.

DDS is an Object Management Group (OMG) standard defined in a language and platform neutral manner. OMG defines standard platform specific mappings to create language specific bindings. Therefore, standardized DDS APIs are available for all OMG supported programming languages; C, C++, and Java being the most popular. Since DDS is a standard in C, C++, and Java, it is available on a wide variety of platforms, including popular real-time operating systems (RTOS), desktop and server operating systems, and Java platforms.

Interoperability

JMS is an API only standard, and does not define an *on-the-wire* interoperability protocol. JMS only requires limited message portability on the client side: a ***Message*** created by provider A should be usable with provider B. Beyond this, a producer written using provider A cannot be expected to deliver messages to a consumer written using provider B.

Currently DDS is also an API only standard. However, there is active progress being made at the OMG towards standardizing on a DDS *on-the-wire* interoperability protocol.

Transports

JMS being an API only specification does not specify a transport model. However, since JMS message delivery is reliable (messages are not dropped unless the provider fails) and ordered, a JMS implementation can benefit from a reliable transport such as TCP. Being connection-oriented, TCP also fits naturally into the client/provider scheme. Centralized server based JMS implementations generally use TCP. They rely on TCP to guarantee reliable and ordered delivery required by the JMS APIs. Some UDP based implementations do exist; they implement the reliable and ordered message delivery semantics on top of UDP.

Like JMS, DDS being an API only specification does not specify a transport model. However, DDS does not depend on reliable and ordered delivery of messages. In fact, the “best-effort” delivery QosPolicy is naturally suited to an unreliable low-latency transport such as UDP, whereas “reliable” delivery may benefit from the use of a reliable transport like TCP. However, since DDS middleware must support both delivery schemes, it cannot make any assumptions about the reliability properties of the underlying transport. Therefore, DDS middleware is less reliant on the capabilities provided by a particular class of transport, and may be able to work well with a variety of transport classes. As an example, the RTI Data Distribution Service implementation provides a *pluggable transport* architecture, wherein any kind of transport can be plugged in, including UDP, TCP, shared memory, and various specialized transports.

Security

JMS has a provision for optionally specifying a *(usernme, password)* when creating a ***Connection***. Beyond this, security issues are left up to the JMS middleware vendor and the client application.

DDS provides an extension mechanism that can be useful in creating secure applications. For example, a consumer application can use a DataReader’s USER_DATA QosPolicy to present security credentials to a DataWriter in the producer application. The producer application can authenticate the security credentials; these may potentially contain authorization rights. If the consumer’s security credentials are not acceptable, the DataReader entity can be permanently ignored using the ***DomainParticipant.ignore_subscription()*** method. A secure transport provided by the middleware vendor can ensure that the data is transferred securely.

Administration

JMS implementations minimally require the use of an external means for configuring and administering ***Destinations*** and ***ConnectionFactories.*** It is common practice to use JNDI to access these objects. Vendor provided proprietary tools must be used to configure the JNDI registries, before they can be used by an application. Evolving an application over its lifetime requires coordination with JNDI administration.

DDS implementations are expected to spontaneously discover each other; the DDS APIs do not rely on external means for establishing the discovery information. For example, with RTI Data Distribution Service, the user application only needs to link in a middleware library; no additional configuration is required to discover and establish dataflows with peer applications. As a result, DDS applications are plug-n-play, and require “zero” system administration.

Performance

Middleware performance can be characterized along several aspects including: (1) the end-to-end latency, i.e. the time required to send a message from a producer to a consumer; (2) the throughput, i.e. the maximum amount of data per unit time that can be transferred from a producer to a consumer. While it is impossible to make any specific comments about performance---this can vary significantly from one middleware implementation to another (regardless of the supported APIs)---it is possible to make some general observations regarding *potential* middleware performance, based on the different choices made by the JMS vs. DDS APIs.

As compared with JMS, DDS has several features that can potentially minimize the end-to-end latency. These include: (1) a “best-efforts” delivery mode that does not require acknowledgements; (2) reduced message overhead, since meta-data such as message headers and properties are not specified per message, but rather on a per endpoint basis; (3) ability to use arbitrary data types which eliminates the need for converting back-and-forth between user and middleware provided types; (4) notification of data availability does not include the actual data, thus avoiding the overhead in setting this up; (5) ability to support “zero-copy” data access so that an application can access the received data directly in the middleware internal buffers without requiring a copy; (6) direct peer-to-peer data transfer from a DataWriter to DataReader without needed an intermediary. Thus, DDS middleware can potentially have better (lower) latency performance,

As compared with JMS, DDS has several features that can potentially maximize the throughput. These include: (a) reduced message overhead as in (2) above; (b) reduced processing in the data path as a result of (3) and (4) above; (c) direct data transfer as in (6) above, which eliminates a potential resource bottleneck and loading point. Thus, DDS middleware can potentially have better (higher) throughput performance as well.

Independent studies have observed DDS implementations that provide a factor of ten performance improvement of over JMS implementations.

Scalability

Scalability refers to the ability to maintain performance levels as more nodes are added to a distributed system. For example, publish-subscribe scales better compared to “remote-procedure-calls (RPC)”, due to the loose coupling between participants. As with performance, it is impossible to make any specific comments about scalability---this can vary significantly from one middleware implementation to another (regardless of the supported APIs). We some general observations regarding *potential* middleware scalability, based on the different choices made by the JMS vs. DDS APIs.

As compared with DDS, JMS has certain features that can potentially limit its scalability compared to DDS. These include (1) PtP messaging domain, which specifies that a message be delivered to exactly one consumer, and behaves as if the messages are put in a single logical queue and handed over to one of the available consumers; (2) centralized server-based architectures, generally used by JMS implementations will be less scalable than decentralized peer-to-peer architectures that support direct data transfer between endpoints.

Real-time applications

DDS and JMS vary in their support for the needs of real-time applications. DDS has a variety of features that directly meet the needs of real-time applications, and have no counterparts in JMS. This is not surprising since DDS was developed while keeping real-time requirements in mind.

The real-time *specific* features of DDS include: (1) a low-latency best-efforts delivery mechanism; (2) qos policies for predictable delivery; (3) qos policies for resource management; (4) status notifications; and (5) potential for lower latency and higher throughput as discussed earlier. In addition, the availability of DDS on high-performance RTOSes and the ability to utilize low latency transports (for example UDP instead of TCP) can further minimize end-to-end latency and support predictable operation. Combined together, they make possible DDS implementations that enable high-performance *real-time distributed applications*.

Enterprise applications

DDS and JMS vary in their support for the needs of enterprise applications. JMS has a variety of features that directly meet the needs of enterprise applications, and have no counterparts in DDS. This is not surprising since JMS was originally developed to provide a Java adaptor for a variety of enterprise messaging middleware implementations.

The enterprise *specific* features of JMS include: (1) full transaction support; and (2) explicit user application message acknowledgements. Combining message acknowledgements with persistent and durable delivery allows enterprise applications to *guarantee* message delivery.

In addition, Java EE, widely used in enterprise environments, supports a M*essage-driven Bean*. A message-driven bean is a data consumer integrated into the enterprise java beans (EJB) framework. Producers are written directly using the messaging API. While the message-driven bean specification does not assume the use of JMS, it is the most commonly messaging technology supported by Java EE vendors.

JMS implementations generally also support for JTA, so that a message application can participate in a distributed transaction. Also, the JNDI APIs generally used by JMS applications are included in the Java EE specifications.

Thus, JMS is well integrated into enterprise application frameworks; given its legacy this is hardly surprising. However, DDS can also be used in enterprise environments; a message-driven bean using DDS can potentially simplify Java EE integration.

Using DDS and JMS together

We have compared the DDS and JMS technologies side-by-side, and considered the practical issues faced when using them. It should be obvious that the choice of DDS or JMS as the middleware technology has a significant impact on a data-centric design. While DDS and JMS offer some capabilities that are similar, there also offer some unique capabilities. Thus, a data-centric design may employ both in complementary ways. There is nothing precluding the use of JMS and DDS together in the same application. In developing a distributed system using both JMS and DDS, certain capabilities can facilitate development and integration. These include (1) DDS-JMS bridging; (2) JMS/DDS bindings; and (3) DDS for JMS discovery.

JMS-DDS bridging

DDS-JMS bridging involves creating a “bridge” that is both a DDS and JMS application. A bridge allows JMS and DDS applications to interoperate.

A bridge forwards JMS messages as DDS data updates, and DDS data updates as JMS messages. A “bridge configuration” file can specify the mapping between DDS and JMS topics, types, and QoS. Such a bridge will incur data conversion and mapping overhead, and introduce a single point of failure between the DDS and JMS domains. It will be limited to supporting the “least common denominator” i.e. only the overlapping capabilities of DDS and JMS. It may not be always possible to provide end-to-end data delivery semantics. However, it can be useful for integrating disparate sub-systems using JMS or DDS.

JMS/DDS bindings

JMS/DDS bindings wrap a DDS middleware with JMS APIs. JMS/DDS bindings can be useful for porting applications written to a JMS API to a DDS domain, or for developing JMS applications that are interoperable with DDS applications, or simply to enable a potentially higher performance JMS implementation.

Since DDS provides finer grained data distribution and management of data flows with many more QoS, it is possible to efficiently map and implement JMS APIs on top of DDS. JMS features not directly provided by DDS, such as PtP messaging semantics, full transactional semantics, and client acknowledgement, can be implemented on top of the DDS APIs. The remaining JMS features can be mapped into DDS APIs.

Note that implementing a DDS API on top of JMS is not a practical idea, since JMS does not provide the fine granularity and low-level primitives needed to provide an efficient DDS implementation.

DDS for JMS discovery

JMS clients rely on non-JMS APIs for creating the ***ConnectionFactory*** and ***Destination*** objects. Typically these objects are obtained by performing a JNDI lookup; these must have been already registered with the JNDI directory.

An application can alternatively utilize DDS to discover these objects. A JMS provider, they could use DDS to announce the configured ***ConnectionFactory*** and ***Destination*** objects to the client applications. A JMS client application would receive the available objects, select the ones it is interested in, and bootstrap the JMS APIs. This approach is useful in a mixed DDS and JMS environment, where DDS and JMS are used simultaneously to distribute different types of information. JMS applications can benefit from the use of DDS's spontaneous discovery mechanism.

Conclusions

DDS and JMS differ in their ability to cater to the key data-centric design requirements. We discussed these differences with respect to (1) data modeling and manipulation, including lifecycle management, data filtering, and transformation; (2) dataflow routing and discovery, including point to point connectivity; (3) delivery quality of service (QoS) per data flow, including delivery effort levels, timing control, ordering control, time-to-live, and message priority; (4) resource specification and management, including resource limits, and history;(5) resiliency to failures, including redundancy and failover, and status notifications; and (6) performance and scalability.

DDS is newer standard based on fundamentally different paradigms than JMS, with regards to data modeling, dataflow routing, discovery, and data typing; these differences enable applications designers with powerful new architectural possibilities. Despite these differences, the user experience of writing to DDS APIs is similar to that of JMS APIs. DDS offers several enhanced capabilities with respect to data filtering and transformation, connectivity monitoring, redundancy and replication, and delivery effort. DDS offers new capabilities with respect to data-object lifecycle management, predictable delivery, delivery ordering, transport priority, resource management, and status notifications. JMS offers some capabilities not offered by DDS. These include client application acknowledgements, full transaction support, message priority, and point-to-point semantics requiring a message to be delivered to exactly one of many consumers.

DDS is amenable to a decentralized peer-to-peer architecture, which can be more robust and efficient compared to centralized server based architecture commonly used for JMS. Unlike JMS, which is a Java language standard, standard DDS APIs are available in many languages. Neither DDS nor JMS provide an interoperability protocol, although there is one currently under standardization for DDS. Neither specifies a transport model, although there are some capabilities in DDS that are better suited to unreliable transports such as UDP, while JMS can generally benefit from the availability of a reliable transport like TCP. Both DDS and JMS defer security to the application, and only provide support for communicating security credentials. Unlike DDS, JMS requires administration of the JMS provider (server) and JNDI registries. The API design choices made by DDS can support potentially higher performance (lower latency and higher throughput) and better scalability than JMS. DDS has some capabilities optimized for real-time applications, not found in JMS. JMS has some capabilities optimized for enterprise applications, not found in DDS.

DDS and JMS can be used simultaneously in an application. Infrastructure already invested in JMS can leverage DDS, and vice-versa. Possible approaches include: JMS-DDS bridging, JMS/DDS bindings, and using DDS for JMS discovery.

If you are designing or integrating distributed data-centric applications, DDS and JMS merit careful consideration. Using one or both can considerably simplify a data-centric design and integration, and help maintain the focus on application issues, rather than becoming bogged down by communication and data delivery concerns.

References

Data Distribution Service for Real-time Systems, v1.1, http://www.omg.org/technology/documents/formal/data_distribution.htm

J2EE Java Message Service (JMS), http://java.sun.com/products/jms/

RTI Data Distribution Service, http://www.rti.com/products/data_distribution/index.html

Acronyms

Acronym	Description
API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
DDS	Data Distribution Service
DCPS	Data Centric Publish Subscribe
DLRL	Data Local Reconstruction Layer
EJB	Enterprise Java Beans
HLA	High Level Architecture
JDBC	Java Database Connectivity
JMS	Java Message Service
JNDI	Java Naming and Directory Service
Java EE	Java Enterprise Edition (previously known as J2EE)
JTA	Java Transaction API
MOM	Message Oriented Middleware
OMG	Object Management Group
P-S	Publish Subscribe
PtP	Point-to-Point
Pub/Sub	Publish Subscribe
RTI	Real-Time Innovations
RTOS	Real-Time Operating System
SOA	Service Oriented Architecture
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language