WHITEPAPER

Data-Centric Programming Best Practices:

Using DDS to Integrate Real-World Systems

Abstract

Systems are often implemented by teams using a variety of technologies, programming languages, and operating systems. Integrating and evolving these systems becomes complex. Traditional approaches rely on low-level messaging technologies, delegating much of the message interpretation and information management services to application logic. This complicates system integration because different applications could use inconsistent interpretations and implementations of information-management services, such as detecting component presence, state management, reliability and availability of the information, handling of component failures, etc.

Integrating modern systems requires a new, modular network-centric approach that avoids these historic problems by relying on standard APIs and protocols that provide stronger information-management services.

For example, many of these systems are heterogeneous, mixing a variety of computer hardware, operating systems, and programming languages. Developers often use Java, .NET, or web-scripting to develop consoles and other GUI-oriented applications, and C or C++ for specialized hardware, device drivers, and performance- or time-critical applications. The end system might mix computers running Windows, Linux, and other operating systems, such as Mac OS X, Android, or real-time operating systems like VxWorks and INTEGRITY. The use of standard APIs and interoperable protocols allows all these systems to be easily integrated and deployed.

Today, these systems are typically developed using a service-oriented approach and integrated using standards-based middleware APIs such as DDS, JMS, and CORBA, and protocols such as DDS-RTPS, Web-Services/SOAP, REST/HTTP, AMQP, and CORBA/IIOP.

This whitepaper focuses on “real-world” systems, that is, systems that interact with the external physical world and must live within the constraints imposed by real-world physics. Good examples include air-traffic control systems, real-time stock trading, command and control (C2) systems, unmanned vehicles, robotic and vetronics, and Supervisory Control and Data Acquisition (SCADA) systems.

More and more these “real-world” systems are integrated using a Data-Centric Publish-Subscribe approach, specifically the programming model defined by the Object Management Group (OMG) Data Distribution Service (DDS) specification.

This whitepaper describes the basic characteristics of real-world systems programming, reasons why DDS is the best standard middleware technology to use to integrate these systems, and a set of “best practices” guidelines that should be applied when using DDS to implement these systems.

Real-World Systems Programming

Real-World systems refer to a class of software systems that operate continuously and interact directly with real-world objects, such as aircraft, trains, stock transactions, weapons, robotic and manufacturing equipment, etc. Unlike systems involving only humans and computers, real-world systems have to live within the constraints imposed by the physics of the external world. Notably, time cannot be slowed, paused, or reversed. The implication is that these systems must be able to handle the information at the pace it arrives at, as well as be robust to changes in the operating environment.

In addition to these environmental considerations, the nature of typical real-world applications also places demands on their availability and need to continue operating even in the presence of partial failures.

In order to interact with the real world, software must include a reasonable, if simplified, model of the external world. This model typically includes aspects of the "state of the world" relevant to system operations. Here the word "state" is used in the normal sense in software modeling and programming. State summarizes the past inputs to the system from its initial state and contains all the information necessary for a system or program to know how it should react to future events or inputs. Imagine that a new component or application starts and joins a system. The "state of the system" contains the information that this new component needs to acquire before it is ready to start performing its function. A typical component would normally only need access to a subset of that state, the portion that directly affects its operation.

For example, in an air-traffic management problem, the relevant aspects of the state of the world might include the current location and trajectory of every aircraft, the flight plans of all flights within a 24-hour window, specific details on each aircraft (type, airline, crew), etc.

Once a software component or subsystem is running, it interacts with other components by exposing part of its state, notifying other components when its state changes, and invoking operations on (or sending messages to) other components. Each component reacts to these information exchanges by updating its internal model of the world and using that to perform its necessary actions.

Defining a Data Model

A data model is simply an organized description of the state of the system. Thus, it includes data types, processes for transferring and updating those types, and methods for accessing the data. It does not typically include functions that can alter the data or (importantly) the application-level logic that affects the data.

Governance organizations and system integrators often start their design by designing the system data-model. There are good reasons for this approach:

- A data model provides governance across disparate teams and organizations, allowing components developed at different points in time by different organizations to be integrated. This makes it an ideal starting point for a central design or governance authority.
- A data model represents the better understood, more invariant aspects of the system. Typically the data model is grounded in the "physics of the system." That is, it describes the kinds of objects and sensors it manages (like aircraft locations, flight plans, and vehicle positions). The data model is not strongly tied to application-specific use cases (e.g., the possible fields in a flight plan are a consequence of the nature of aircraft flight); this makes the data model a good starting point, since the full set of use cases might not be well known in advance or might be the responsibility of a different team.
- A data model increases decoupling between systems and components. The data model is grounded in the essential information present in the system and it does not depend so much on the use cases that access the information. For example, an air-traffic control model might include a definition of a "flight plan," but not whether it is automatically generated using an optimization algorithm, checked for collisions, or altered in mid-flight. Using the data model as the basis for the integration avoids over-constraining the design, leaving it open to allow future evolution and use cases. Contrast this with a design based on defining service invocation APIs which are intimately tied to the details of each service and are likely to change as new use cases are incorporated

Example Data Model

Imagine designing a simple "chat" application. The underlying Data-Model could be defined to contain four kinds of objects summarized in the table below:

Object Kind	Key Fields	Other Fields	Description
Person	EmailAddress	Name, Location, Age, Picture, Avatar	Identifies each individual that can participate in "chat" conversations
Account	EmailAddress	Password, ChallengeQuestion, ChallengeAnswer	Provides the account credentials necessary to authenticate individual persons.
ChatRoom	Name	Description, MembershipList	Defines a ChatRoom and lists the individuals that are participating in the chat
ChatMessage	SenderEmail, Timestamp	Contents, ChatMessageDestination	Contains a chat message sent to a ChatRoom or a Person

The table above provides a very informal description. Normally the Data Model would be described in formal a high-level language such as UML. See below for an example:

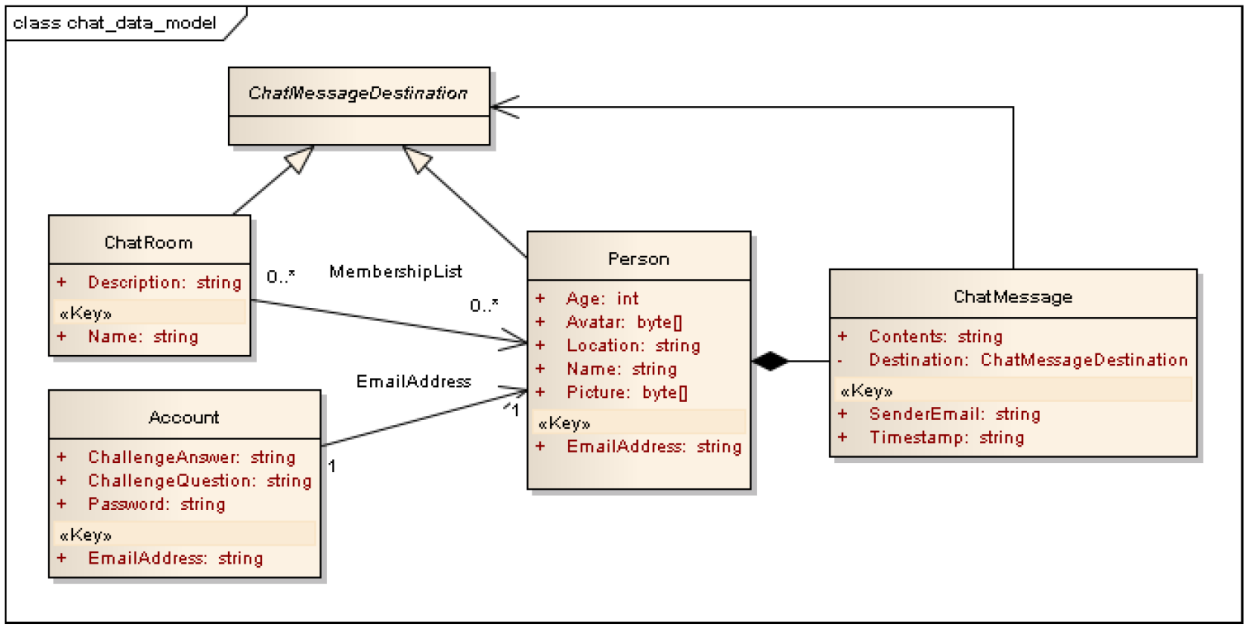

Figure 1.

DDS Maintains the State of the World as Defined by the Data Model

The single most important benefit of using DDS is that its programming model directly supports the expression, maintenance and distribution of the "state of the world" data model. No other standard messaging middleware technology does this.

DDS provides the means to create Global Data Spaces where applications can create or delete data objects and update their state. Each DDS Global Data Space is uniquely identified by an integer (the DDS Domain ID) and is maintained separately from the others. Within a DDS Global Data Space, each data object is uniquely identified by an application-defined string (the DDS Topic Name) and the values of a set of application-defined fields in the data object (the DDS Topic Keys). All data objects belonging to a DDS Topic share a common application-defined schema or data type that can be defined in a variety of languages, such as IDL, XSD, or XML.

With DDS as the underlying middleware, as soon as you have a data model, you also have a direct way to implement it in a distributed system and access it from your applications. This is because DDS allows direct mapping and access to the data model. All the governance body or system architects need to do is materialize the data model in terms of a type-definition language (like XSD, XML, or IDL), define the separate Global Data Spaces that should be used, and map each of the separate types of data objects to a DDS Topic name. This process is simple, unambiguous, and can be done using standard languages as part of the data model definition—without requiring any glue code or application-specific mappings.

Once the data model is mapped to DDS domains, Topics and Keys, the APIs to interact with the data model are already given. They are the standard CRUD (Create, Read, Update, and Delete) operations which can be applied to any data object in the DDS Global Data Space.

As an added bonus, you can attach DDS Quality of Service (QoS) policies to your data model. QoS policies allow you to specify things like whether some collections of data objects should be sent reliably (RELIABILITY policy), whether a specific collection of data objects (e.g., those representing the current position of all UAVs currently flying) should be updated at a specific rate (DEADLINE policy), the relative priority in notifying and providing updated values when a data object changes (LATENCY_BUDGET policy), whether the state of specific sets of data objects should be kept in durable storage and made available to new components when they appear in the system (DURABILITY policy), etc.

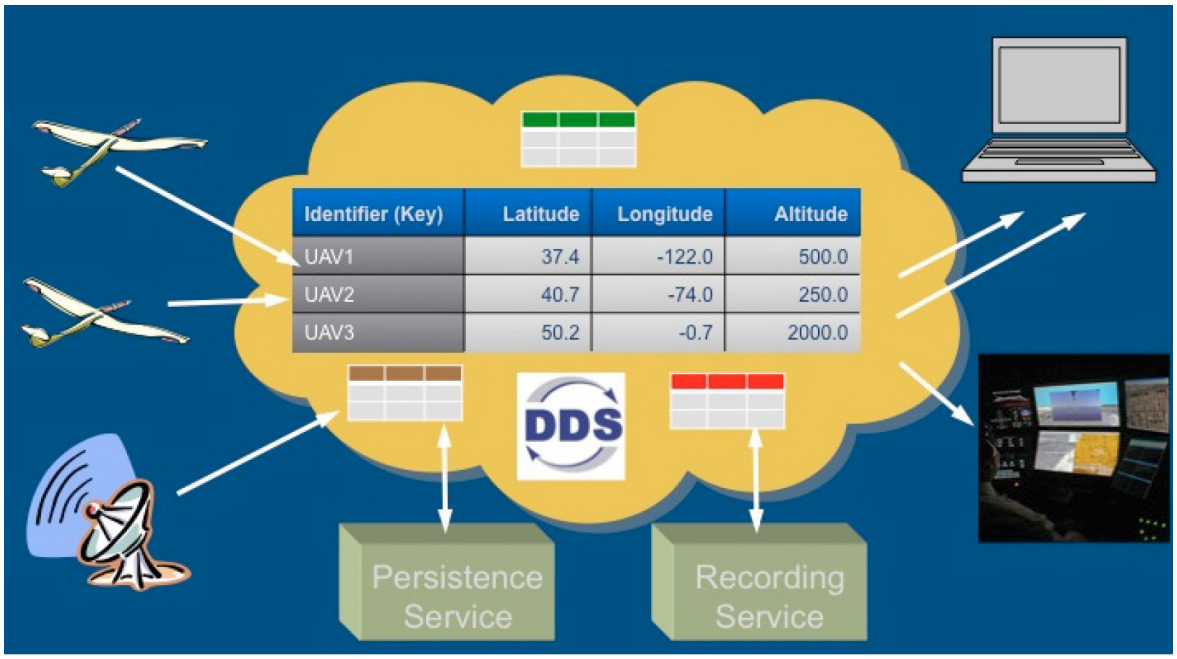

Figure 2. Example system that uses DDS to share the state of a set of UAVs. A Global Data Space is defined to contain all the information relevant to the components of the system. A DDS Topic (e.g., named "UAV Location") has been defined with an associated schema containing the UAV's identifier (Key), and its latitude, longitude, and altitude. Other DDS Topics, represented by the green, brown, and red tables, are also part of this same DDS Global Data Space.

Finally, DDS has many built-in services that provide all the "state management" features that normally would have to be implemented in application code. For example, DDS maintains the "history" of changes for any data object up to a configured "depth" per Topic. The history can be maintained in terms of number of changes or in terms of the timestamp when the change was made (HISTORY and LIFESPAN policies). DDS can also arbitrate among redundant sources of data, providing ways to set up reliable systems with failover schemas. It can monitor the presence and liveliness of applications and then notify the user when writers of a data object drop out. DDS allows applications to filter updates based on their frequency or content, thus saving bandwidth and processing. It automatically discovers new applications and Topics as they appear. It can record changes, store

data in databases, allow you to monitor and visualize the data, and more. All these capabilities are provided by standard components and APIs, saving users considerable application code, as well as the associated test code, documentation, and maintenance.

About DDS

DDS stands for Data Distribution Service. It is a set of specifications standardized by the Object Management Group (OMG). OMG (www.omg.org) is the largest systems software international standards organization, known for many specifications including the Unified Modeling Language (UML). The current DDS family of specifications is summarized in the table below:

Date Distribution Service for Real-Time Systems (DDS)	Specification of the programming model, QoS, and language APIs used to program a Data-Centric Publish-Subscribe application.
Data Distribution Service Real-Time Publish-Subscribe Interoperability Wire Protocol (DDS-RTPS)	Specification of the wire protocol used by DDS to exchange information. It includes discovery, data encapsulation, reliability, multicast, and many selectable QoS parameters.
Extensible and Dynamic Types for DDS (DDS-XTYPES)	Specification of the valid set of data types that can be sent/received via DDS, as well as how to describe them using languages like IDL, XSD, or XML; how to represent them in a serialized form when they are sent over the wire; and how to access the data and types from a programming language using plain language objects or dynamic APIs.
UML Profile for DDS (DDS-UML Profile)	Specification of how to model a DDS system using a UML tool.
DDS for Lightweight CORBA Component Model (LwCCM)	Configuration of DDS QoS via profiles defined in XML, and extensions to the LwCCM specification so it can leverage DDS.

In addition to enabling the data-centric approach, DDS has these advantages:

- Existence of solid, multi-vendor supported standards for both APIs and protocols
- Messaging performance and scalability
- Ability to integrate different operating systems and programming languages
- Configurability via QoS

Standards are important to reduce costs, avoid vendor lock-in, and ensure the long-term availability of the technology. The DDS family of standards is the only one that covers the programming language APIs (ensuring portability between implementations), the wire protocol (ensuring interoperability between components that use different implementations of the middleware) and QoS. And it does so for multiple programming languages, such as C, C++, Java, .NET, Ada, etc. Competing standards like JMS or Web-Services lack one or more of these aspects.

Performance and scalability are often critical in “real-world” event-driven systems. Since you cannot slow down or stop time, systems must be able to handle events at the rate they occur or else fail with potentially costly or harmful consequences. Independent tests have demonstrated that DDS implementations provide the highest performance standards-based middleware available. Features like a protocol that natively supports reliable multicast and operates without brokers or servers make DDS stand out when compared to alternatives like JMS or Enterprise Service Bus (ESB) implementations.

Many real-world systems are heterogeneous: they mix a variety of hardware and software platforms. It is not uncommon to see graphical interfaces built using Java or .NET technologies, with performance-critical components in C or C++, running on a mix of Linux, Windows, and real-time operating systems, and connected over various transports. These systems greatly benefit from a standard API that can be used in many programming languages and is supported on a wide variety of platforms and transports. DDS uniquely provides that capability.

Configurability via QoS can be extremely important for components that have limited capacity due to size, weight, power, or deployment considerations. A typical real-world system might mix in some small hand-held computers or computers that can only communicate over low-bandwidth networks. While this might not be the case for every component, it is important that the ones with these limitations can still participate in a system without being overwhelmed by the information the rest provide, and also without slowing down the rest. Configurability via QoS is the key here; it is natively supported by the DDS standard's built-in library of 20+ QoS policies. This is also important when a system must accommodate some operating change or partial failure condition; in that case, QoS allows the more important information to be properly prioritized and managed.

Best Practices in DDS Programming

As we have seen, DDS is a powerful tool to integrate components and systems. We now have extensive experience in hundreds of applications, and can recommend these guidelines to best realize its potential:

- Start by defining a data model, then map the data-model to DDS domains, data types and Topics.
- Fully define your DDS Types; do not rely on opaque bytes or other custom encapsulations.
- Isolate subsystems into DDS Domains. Use mediation, such as RTI Routing Service, to bridge Domains.
- Use keyed Topics. For each data type, indicate the fields that uniquely identify the data object.
- Large teams should create a targeted application platform with system-wide QoS profiles and limited access to the DDS APIs.
- Configure QoS using XML Profiles.

The rest of this whitepaper further details these guidelines.

Start by defining a data model, then map the data model to DDS domains, data types and Topics

From the previous discussion, it should be clear why architects should start by defining a data model. The point of this guideline is to encourage people not to skip this step, and to complete it by defining how the data model should be mapped to the DDS Global Data Space.

Use the data model to express the essential information that defines the state of your system. Think about the reactions you expect in the system to the state changes and use this to check that your data model is complete.

It is often useful to divide the data model into a control plane and a data plane. This allows separation of the application and system management functions from the logic of the application components them-selves.

A checklist for a well-defined model:

- It can be fully expressed in terms of data structures and their relationships.
- It can be easily explained and understood to someone familiar with the problem domain, without requiring the person to be knowledgeable in programming or software engineering.
- It is defined using a platform-independent modeling language, such as UML, XML or IDL. This way the model is not tied to a specific platform, technology, or deployment scenario. The same model could be deployed on top of different technology stacks (DDS, WSDL, message bus), programming languages (Java, C++, C#), and configurations (platforms, networks, etc.).
- There are no location dependencies in the model. It can be deployed on a single computer or on a network.
- The different scenarios and behaviors of the system can be expressed in terms of changes to the state of elements in the data model.

Following this guidance will result in a well-defined data model and an unambiguous, portable, and interoperable way for applications to communicate according to the data model. This will ensure your system remains open to future changes, allowing new components and use cases to be added with no need to alter the existing ones.

Example

For the simple ChatRoom data-model defined earlier, we could use a single DDS Domain and data types that correspond to each kind of object: Person, Account, ChatMessage, and ChatRoom.

A simple mapping of the data model to DDS would use separate Topics for each ChatRoom. The name of the Topic could be built by adding the suffix "_ChatRoom" to the name of the ChatRoom. For example for a ChatRoom with name "DDS_News" we could use the Topic name "DDS_News_ChatRoom". Similarly, we could use separate Topics for each Person. The name of the Topic could be constructed by adding the suffix "Person" to the email of the Person. All these Topics would be associated with the "ChatMessage" data type. In the mapping to DDS, we would only use the "SenderEmail" field as the DDS Key. This would allow us to better control the history of messages cached by an application.

Given this mapping, the creation of a new ChatRoom (e.g. "DDS_News") corresponds to creating the corresponding DDS Topic ("DDS_News_ChatRoom"). Joining a ChatRoom corresponds to creating a DataWriter (so we can send messages) and a DataReader (so we can receive messages) on that Topic. Writing to the ChatRoom "DDS_News" would correspond to writing a ChatMessage object to the corresponding Topic "DDS_News_ChatRoom". Sending and receiving messages to a Person uses a very similar approach based on the Topics that use the "_Person" suffix.

Additional Topics could be introduced to monitor and manage the users and their accounts.

Other mappings are also possible. For example, all ChatRoom messages could share a single Topic and the messages destined to a specific ChatRoom could be selected by using a content filter on the Destination field of the ChatMessage. A description of the benefits of each approach is beyond the scope of this paper.

Fully define your DDS Types, do not rely on opaque bytes or other custom encapsulations

Some architects and integrators map the data model to DDS domains and Topics without using the type-definition facility available in DDS. Instead, they abuse the DDS "opaque bytes" or "string" types and manage all the data marshalling and demarshalling at the application layer.

Mapping the data model to opaque bytes or strings (even XML strings) is a bad practice for several reasons, some obvious and some subtle:

Using opaque bytes or strings requires more application code to be written, tested, and maintained. This also means there must be a separately maintained document that explains the marshalling. Moreover, the code to marshal and demarshal must be present in all application components, meaning that either it has to be physically shared—a hard task when components are implemented over time or by separate vendors—or it has to be duplicated by each team and in each programming language used. These steps create opportunities for error and cost.

Perhaps less obvious is that using opaque bytes or strings will prevent the middleware from giving you a lot of "free services." For example, the middleware will not be able to perform content filtering for you. Not only does that mean that the application must do the filtering, but it is also far less efficient on the network, since filtering will have to be implemented on the receiver side. When DDS is aware of the data type and filtering requirement, it can filter at the source. This avoids sending the data to subscribers that are not interested, saving valuable CPU and network bandwidth.

Many other built-in DDS services depend on having access to the data type: routing and mediation, storing data into relational databases, exposing data to web clients using HTTP, visualizing data in tools, and integrating with Microsoft Excel are just a few. One of the most powerful features of DDS is content awareness; using opaque types throws this away.

Note that exposing types to DDS does not mean you must propagate them via discovery. This is the default, but can be separately configured. Exposing the type simply means there is a well-defined language (defined by the OMG and the W3C) that describes the data types and ways to marshal that type into network messages.

In summary, if the data exchanged at the system interfaces is not strongly typed, you very likely have an integration problem waiting to happen. Use of opaque data types should be the exception. A strongly typed interface makes it possible for the middleware to intelligently filter data at the source, somewhere in the middle, or at the end; generic transformation rules can be applied; new components can easily be integrated; etc.

Example

In our ChatRoom example, we defined the data type associated with the ChatMessage. As a consequence we can now use content filters to select messages that contain certain text or are sent by people of certain age ranges. We can record and automatically store ChatMessages in a database so we can run queries that monitor certain patterns; we could later add more services (e.g., a translation service) that use the content, etc.

Isolate subsystems into DDS Domains. Use mediation, such as RTI Routing Service, to bridge Domains

Large systems are best developed as a composition of smaller subsystems. Subsystems are often developed by different teams or even different companies, and are subject to different governance and conventions.

Rather than considering the data model as a single monolith, it is often better to think of it hierarchically. At the top level is the information that needs to be shared among the top-level subsystems; this must be agreed upon and controlled using a process that takes into consideration the needs (and possibly involves the stakeholders from) all these subsystems. Once this is done, each subsystem can separately define the information that will be shared among components in the subsystem (but not between subsystems). This information can be defined using a process that takes into consideration only the needs of the subsystem and can be subject to different governance and revision cycles. Complex subsystems can be further broken into other subsystems, recursively applying the previous process.

The mapping of the data model into DDS should preserve this modularity. The system should use DDS Domains to isolate subsystems that have little overlap in the information model, are developed by different communities, or have significant size or complexity:

If two subsystems have little information model overlap, then the processes/components in one subsystem are primarily sharing information with other processes/components in that same subsystem. By placing the subsystems in different DDS domains, the traffic will be completely separate, and can use different multicast addresses and ports. Moreover, discovery information will also be kept separate so that processes and applications are not made needlessly aware of other processes/applications (from the DDS discovery point of view). This could save significant network bandwidth, CPU, and memory on each of the participating components.

If two subsystems are developed by different communities, they will likely follow different governance and will want to place explicit controls on the information that enters and exits the subsystems. Often the information that traverses system boundaries will need to be mediated, changed in format, pruned of certain fields, etc. It is also common for these subsystems to be deployed in separate network segments or be protected by firewalls. In all these cases, placing each subsystem in a separate domain provides a natural way to control the scope of the information and an easy ways to manage the information that flows in and out of the subsystem. Turnkey bridging and mediation applications, such as RTI Routing Service, can be used to easily administer how the information flows, enable and disable flows, inject transformations, and monitor flows.

Finally, some subsystems are large or complex, and involve many hundreds or thousands of processes communicating over DDS, with hundreds of DDS Topics. These should be partitioned into separate DDS Domains. The partitions should be chosen so that information shared among processes in the same partition is maximized, and information flowing between partitions is minimized.

Example

Rather than exposing all ChatRooms to all users, we could use DDS Domains to create separate, shared areas. Only people in a specific Domain would see and have access to the ChatRooms in that Domain. If certain messages need to traverse Domains, we can configure DDS routing services to accomplish this goal. This approach can be used to scale the number of users and ChatRooms to very large numbers.

Use keyed Topics. For each data type, indicate to DDS the fields that uniquely identify the data-object

Sometimes integrators mapping their data models to DDS Types do not indicate which fields within their data types uniquely identify each data object. That is, they do not define any DDS Keys.

While there are some valid scenarios in which a DDS Topic should not have a key, these situations are quite rare. As a rule of thumb, all DDS Topics should have keys. A similar situation occurs when defining database schemas. How often do you encounter SQL tables that have no key fields? Without a key, you would only be able to have a single record (row) in a table, or there would be no concept of updating a record, only inserting new records, resulting in ever growing tables. Either way, there would be no way to store "state" into tables.

Additionally, many DDS QoS policies and built-in services depend on having a key:

- **History cache**. DDS can keep a cache of the last set of changes (e.g., say the last 10 changes) applied to each data object. This is done separately for each data object, so a rapidly changing object does not "replace" the last value of another one that changes less frequently. This capability can also be maintained by the source application and the durability service that initializes late-joining readers. This feature is also very important for late-joining readers that want to be initialized with the most current value of each object (or the last few changes that happened to each data object). Without a key, DDS cannot perform smart caching and late joiners will be forced to replay through old history before getting to the most current value.
- **Ensuring regular data-object updates**. DDS contains a built-in mechanism that ensures each data object is updated regularly. The configuration is made per Topic using the DEADLINE QoS policy. If a data object is not updated, the application is immediately notified with the ID of the offending object. Statistical counters are also maintained, allowing external applications to monitor the health of the system. If a Key is not specified, this facility will not be available.
- **Ownership arbitration and failover management.** DDS applications can use the OWNERSHIP QoS policy to specify that data objects can only be updated exclusively by one writer. The owner of each data object is automatically managed by DDS based on the presence of data writers, their ownership strength value, and their ability to meet the QoS they signed up for. DDS will automatically fail over to the highest-strength active data-writer.
- **High-performance, content-based filtering on multicast networks.** RTI Data Distribution Service has the capability to leverage the smart multicast-filtering features available in most network switches. This feature can greatly increase the data-distribution scalability and performance in situations where a single DDS Topic has many subscribers, each needing only a subset of the information published on the Topic. If the Topic has defined a set of key fields and the subscribers are selecting the data of interest based only the value of the key fields, then the filtering can be done with extreme efficiency and virtually no impact on the publisher.

- **Integration with relational databases.** Data distributed over DDS often originates or terminates in relational database tables. Individual records in a table are uniquely identified by the values of the fields marked as the "key" for that table. If those same fields are also marked as forming the key for the corresponding DDS Topic, then the integration is seamless and the DDS data cache can work hand-in-hand with the database table storage. If the DDS Topic does not define a key, then DDS cannot take advantage of several optimizations which are especially important in situations where subsystems can become disconnected or have bursty traffic.
- **Visualization in Excel and other table-oriented displays.** Understanding and monitoring a system often require visualization of the data in the DDS Global Data Space, as well as the data sent over DDS. It is very natural to visualize this information using tabular displays, where each row represents the value of a specific data object. These displays can be updated live as applications publish updates to these objects, record object creations and deletions, provide historical views of the evolution of each data object, etc. The DDS standard's data-centric features make it possible to create generic displays that will work for any kind of data and therefore require no programming. With RTI's Spreadsheet Add-in for Microsoft Excel, you can look at this data live.
- **Smart management of slow consumers and applications that become temporarily disconnected.** In real deployment scenarios, components or subsystems can become temporarily disconnected and rejoin later. Some components may be slower or become busy and unable to keep up with the information they requested; the information volume might increase in a bursty or unexpected manner, etc. Given that time cannot be slowed or stopped, real-world systems must find a way to handle these situations gracefully. One option is to buffer messages, hoping that when the situation improves the system will be able to catch up. However, this is not always feasible without exceeding some internal resource limit. Even if memory is unlimited, it is not always desirable to save all that old data. When the situation recovers, a buffer full of old data will burden the lagging consumer with old information, when it should be reacting to the most current. This might be wasteful, expensive, or even fatal. If, on the other hand, the DDS Topic contains a key, then DDS can be smart and only cache and deliver the latest updates to each data object.
- **Achieving consistency among observers of the Global Data Space.** There are many scenarios where DDS is used to communicate and synchronize portions of a system's state. An example of this state could be the current arrival and departure times of aircraft, the current location of UAVs being controlled by a set of ground stations, or the current classification of a set of radar tracks. Many of these scenarios can have different sources contributing to different portions of the global state. Building a consistent state picture in these scenarios is a very challenging problem; in the past, it required very complex application code. DDS can handle this problem via QoS policies such as DESTINATION_ORDER and OWNERSHIP. Using DDS mechanisms, even if the state observed by different consumers is temporarily different due to timing or temporary message loss, DDS will ensure it eventually converges. However, this will only work correctly if the application has specified keys for the Topic; otherwise there are situations where portions of the state could be lost.

There are Very Few Reasons not to Use Keys

In our experience, people do not define keys because (1) they do not understand the implications, (2) they are concerned that using them would "lock" them into using DDS, as keys are not supported by other publish-subscribe middleware like JMS, or (3) they are afraid it will impact performance or resource usage.

Hopefully, the importance and value of keys are now clear. Avoiding keys for "portability" to other publish-subscribe middleware technologies is misguided. Rather than leveraging what is already in DDS, this capability will need to be re-implemented at the application level. This incurs obvious costs. The application also cannot really do this job correctly because it has no efficient way to control the middleware's internal queues and protocol state. Application-level implementations usually must introduce extraneous brokers or mediation components, which themselves introduce new bottlenecks and failure modes.

Performance is not a significant concern. Extensive benchmarking on RTI Data Distribution Service shows that in almost all scenarios, the performance impact introduced by using keys is negligible. Latency differences are typically less than 5%.

In most situations, the extra resources consumed by DDS Keys are very small and typically less than what the application would otherwise use itself to implement the equivalent functionality. However, in some cases with millions of instances, it might not be practical to have DDS manage the instances and it is better not to define a key.

We believe that the only real reason not to define a key is when the Topic represents a pure message and there is a natural way to view it as an update to a data object. This situation is rare. Imagine an instant-messaging system where users or agents exchange messages (or a typical phone SMS system). While the messages do not strictly represent updates to data objects, it is still very helpful to think of them as such, with the key being, for example, the identity of the message sender. This allows receivers to organize and view messages according to the sender, cache the last "N" from each sender, efficiently filter by sender, etc., without writing application code.

Example

In the ChatRoom example, we defined keys for every data type. Specifically, we used the "SenderEmail" as the key for the ChatMessage data type that we use to send chat messages. This selection allows applications to configure the number of messages that should be preserved from each user on each ChatRoom. This count can be selected by setting the "history depth" attribute in the HISTORY QoS Policy.

Large teams should create a targeted application platform with system-wide QoS profiles and limited access to the DDS APIs.

When DDS is used in projects that involve large teams, a small "infrastructure" team should provide a simplified interface to the underlying middleware (and operating system) services. This team should provide XML-based QoS profiles. The platform should tailor the DDS API to the application. However, do not deeply wrap the DDS APIs. Rather, provide tailored APIs to configure the system, define the types, and create the necessary DDS entities, but allow developers to access the underlying DDS entities so that they have access to the features and standard capabilities available in DDS as their systems evolve.

The wrapper should not end up looking like a mini "pub-sub" API. Instead, the wrapper should provide communications in the application's terms and with standardized QoS profiles. This would preclude un-predictable use of QoS combinations by those less trained in the standard and simplify configuration and programming of simple use cases.

However, teams should not completely wrap the DDS API, which would unnecessarily limit flexibility. Instead, we recommend:

- **Wrap creation, not communication.** Most designs should provide constructor methods that automate the creation of common types. However, there's little reason to wrap the sending and receiving functions. The basic DDS API is already very simple, so the platform can support direct calls to "read" and "write" the data.
- **Provide QoS profile files.** DDS has a large API for dealing with QoS configuration. Infrastructure teams could wrap this API; however, it is simpler, more maintainable, and more flexible to use the facility that allows configuration of QoS via XML files.

- **Expect evolution.** The wrapper API is normally defined at an initial point in the project. Wrapping the entire API requires an arbitrary decision about which DDS features will be "important" or "useful." The team developing the wrapper might have limited knowledge of DDS, might want to take advantage of new features, or might not anticipate some application use-case that could benefit from DDS features. Therefore, important features or QoS policies might not be exposed. At a later time, when this is needed, the presence of the wrapper makes it hard to access these features because a separate module has to be modified and rebuilt. Therefore, the wrapper should provide access to the underlying DDS Entities (DDS DomainParticipants, Topics, Publishers, Subscribers, DataWriters, and DataReaders); that way when extra functionality is required, it will be readily accessible.
- **Carefully add functionality to the wrapper.** Done well, adding application-level functionality to the wrapper can reduce application-level code. Done poorly, this is limiting. Thus, this should be treated as a system-design level decision.

Other situations that could justify development of a wrapper are:

- **When adding functionality or additional protocols that use DDS underneath.** For example, when implementing a client/server (request/reply API) on top of DDS.
- **When the intent is to create a domain-specific API that changes the programming model.** In other words, the wrapper is not intended to be used for sending/receiving messages or reading/writing data. Rather, its purpose deals with something specific to its domain, which under the hood is implemented using DDS. In these situations, the mapping between operations in the wrapper layer and operations in the DDS API might be complex and involve multiple calls, which further justifies the creation of this wrapper.

Example

If we expect many users to program Chat applications, GUIs, etc., then we could define a wrapper API that exposes objects and operations that would be more intuitive for people thinking of "Chat" applications. For example, we could define a ChatRoomClass, so that rather than creating a DDS Topic, the application would construct a ChatRoomClass object (the implementation of the constructor would then be responsible for creating the DDS Topic). Similarly, there would be operations to "join" the ChatRoom, send a ChatMessage, etc. which would wrap the corresponding DDS operations.

Configure QoS using the XML Profiles

The ability to use XML to configure DDS QoS was standardized by OMG in 2008 as part of the DDS for LwCCM specification, and has been available in RTI Data Distribution Service since version 4.4.

This feature allows an application developer or system integrator to define the QoS values that will be used in an XML file. The values of the QoS are not compiled into the executable; therefore, they can be changed each time the application starts. This makes it easy to optimize and tune QoS related issues (performance, scalability, reliability, resource consumption, availability, etc.) at integration or deployment time.

The implementation of this feature makes it very easy to use and manage. For example, a full XML schema document (XSD) is provided and can be used to validate the XML file that defines the profiles. Moreover, many editors will use this XSD to provide auto-completion and help when creating the QoS profile file.

In addition, the QoS profiles defined in XML support profile inheritance, so new profiles can be defined as specializations of existing ones. They also support conditional specification based on the Topic name, so that a single file can be used to separately configure the QoS that will be used for each Topic. They can be organized into libraries, etc. All these features make using XML-based QoS profiles powerful and easy.

Conclusions

Real-world systems must operate continuously and interact directly with real-world objects. They must perform within the constraints and timing imposed by the physical world. In practice, this means they must be able to handle the information as it arrives and be robust to changes in the operating environment.

Such systems are increasingly being integrated using a data-centric, publish-subscribe approach, specifically using the programming model defined by the OMG DDS specification. The main benefit of DDS is its ability to map the application data-model directly into application code. DDS is the only middleware standard that covers the programming language APIs (ensuring portability between implementations), the wire protocol (ensuring interoperability between components that use different implementations of the middleware) and QoS. It also supports multiple programming languages, such as, C, C++, Java, .NET, and Ada.

This whitepaper defined "best practices" guidelines, gleaned from extensive experience with hundreds of DDS based applications. They should be considered when using DDS to implement real-world systems. These guidelines can be summarized as follows:

- Start by defining a data model, then map the data model to DDS domains, data types and Topics.
- Isolate subsystems into DDS Domains. Use mediation, such as RTI Routing Service, to bridge Domains.
- Fully define your DDS Types; do not rely on opaque bytes or other custom encapsulations
- Use keyed Topics. For each data type, indicate to DDS the fields that uniquely identify the data object.
- Large teams should create a targeted application platform with system-wide QoS profiles and limited access to the DDS APIs.
- Configure QoS using the XML Profiles.

About

1. OMG DDS Specification version 1.2. http://www.omg.org/spec/DDS/1.2/PDF/
2. OMG DDS Real-Time Publish-Subscribe Protocol, interoperability Wire Protocol version 2.1. http://www.omg.org/spec/DDSI/2.1/PDF/
3. Extensible Topics for DDS Specification. http://www.omg.org/spec/DDS-XTypes/1.0/Beta1/PDF
4. DDS for LwCCM Specification. http://www.omg.org/spec/dds4ccm/1.0/Beta1/PDF
5. SPAWAR NESI Guidance. http://nesipublic.spawar.navy.mil/nesix/View/P1190
6. RTI Benchmarks. http://www.rti.com/products/dds/benchmarks-cpp-linux.html

About RTI

Real-Time Innovations (RTI) is the Industrial Internet of Things (IIoT) connectivity company. The RTI Connext® databus is a software framework that shares information in real time, making applications work together as one, integrated system. It connects across field, fog and cloud. Its reliability, security, performance and scalability are proven in the most demanding industrial systems. Deployed systems include medical devices and imaging; wind, hydro and solar power; autonomous planes, trains and cars; traffic control; Oil and Gas; robotics, ships and defense.

RTI is the largest vendor of products based on the Object Management Group (OMG) Data Distribution Service™ (DDS) standard. RTI is privately held and headquartered in Sunnyvale, California.

Your systems. Working as one.

CORPORATE HEADQUARTERS
232 E. Java Drive
Sunnyvale, CA 94089

Tel: +1 (408) 990-7400
Fax: +1 (408) 990-7402
info@rti.com

www.rti.com