rti

WHITEPAPER

Is DDS for You?

Abstract

Today's real-time systems are increasingly distributed; they exchange data between many discrete applications in a networked system. Several middleware designs have arisen to meet the resulting messaging and integration requirements, including client-server, message passing, and publish-subscribe architectures.

The Object Management Group (OMG) Data Distribution Service for Real-Time Systems (DDS) standard is the first comprehensive specification available for "publish-subscribe" data-centric integration. This paper helps system architects understand when DDS is the best fit to an application. It first provides an overview of DDS's functionality and compares it to the other available technologies and standards. Then, it examines driving factors, potential use cases, and existing applications. It concludes with general design guidance to help answer the question of when DDS is the best integration solution.

Introduction

Today's complex real-time systems are distributed. Applications like military systems, air traffic control, unmanned vehicles, financial trading, and medical imaging systems comprise many programs running on many network nodes. These systems must coordinate hundreds of computers working together to form a single working system.

The fundamental challenge is to disseminate information quickly to many nodes. It would seem a manageable problem. Networking hardware is fast, cheap, and reliable. The software is also mature; all modern operating systems provide a basic network stack.

However, the stack provides only low-level routing and connection management. How does the system know where to get information? When is the data available? When do other nodes need it? What happens if nodes join or leave the system? How should the system react to failures? What happens when hundreds or thousands of programs interact? While it is simple to write network code to connect a few nodes, these questions quickly overwhelm simple approaches. Complex distributed applications require a more powerful communications model.

Types of Middleware

Several types of distribution middleware technologies have emerged to meet this need. They fall into three broad classes: client-server (or remote object invocation), message passing, and publish-subscribe.

Client-Server

"Client-server" was the buzzword of the exploding IT market of the 80's and 90's. Client-server networks connect servers that store or process data to clients that request and use data. Most client-server middleware designs present an Application Programmer Interface (API) that strives to make the remote node appear to be local; to get data, users call methods on remote objects just as if they were on the local machine (also called Remote Method Invocation, RMI). The middleware thus strives to hide the networked nature of the system. Client-server middleware designs include SOAP, HTTP, CORBA, and Enterprise JavaBeans (EJB).

Client-server middleware technologies typically build on top of TCP, the Transmission Control Protocol. TCP offers reliable delivery, but little control over delivery Quality of Service (QoS). For instance, TCP retries dropped packets, even if the retries take a lot of time. TCP requires dedicated resources for each connection; since each connection keeps track of the state of messages. Thus, it takes time and significant resources to set up and maintain connections. TCP does not scale well for extended data distribution.

Message Passing

Message-passing architectures work by implementing queues of messages as a fundamental design paradigm. Processes can create queues, send messages, and service messages that arrive. This extends the many-to-one client-server design to a more distributed topology. With a simple messaging design, it's much easier to exchange information between many nodes in the system.

Some operating systems use message passing as a fundamental low-level synchronization mechanism. Others provide it as a library service (such as POSIX® message queues). Many enterprise middleware designs implement a message-passing architecture; for example, IBM's WebSphere MQ.

Message passing allows direct peer-to-peer connections. However, message-passing systems do not support a data-centric model. With messages, applications have to find data indirectly by targeting specific sources (by process ID or "channel" or queue name) on specific nodes. The model doesn't address how you know where that process/channel is, what happens if that channel doesn't exist, etc. The application must determine where to get data, where to send it, and when to do the transaction. There is no real model of the data itself, there is only a model of a means to transfer data.

Also, messaging systems rarely allow control over the messaging behaviors or real-time QoS. Messages flow in the system when produced; all streams have similar delivery semantics.

Publish-Subscribe

Publish-subscribe adds a data model to messaging. Publish-subscribe nodes simply "subscribe" to data they need and "publish" information they produce. Messages logically pass directly between the communicating nodes. The fundamental communications model implies both discovery—what data should be sent—and delivery—when and where to send it. This design mirrors time-critical information delivery systems in everyday life including television, radio, magazines, and newspapers. Publish-subscribe systems are good at distributing large quantities of time-critical information quickly, even in the presence of unreliable delivery mechanisms.

Publish-subscribe designs are also not new. Industrial automation "fieldbus" networks have used publish-subscribe designs for decades. Commercial publish-subscribe enterprise solutions (e.g. Tibco Rendezvous) routinely deliver information such as financial data from stock exchanges. In the embedded space, commercial middleware products, including RTI® Data Distribution Service, control Navy ships, traffic grids, flight simulators, military systems, and thousands of other real-world applications. The technology is proven and reliable.

Middleware Summary

In summary, client-server middleware is best for centralized data designs and for systems that are naturally service oriented, such as file servers and transaction systems. They struggle with systems that entail many, often-poorly-defined data paths.

Message passing implements "send that there" semantics, and maps well to systems with clear, simple dataflow needs. They are better than client-server designs at free-form data sharing, but still require the application to discover where data resides.

Publish-subscribe includes both discovery and messaging. Together, they implement a data-centric information distribution system. Nodes communicate simply by sending the data they have and asking for the data they need.

Figure 1. Super-TV sends and receives many channels

What is DDS?

DDS is a standard managed by the Object Management Group (OMG), the systems software standards body responsible for technologies like middleware and modeling languages (e.g. UML). The OMG DDS standard is the first open international standard directly addressing publish-subscribe middleware for real-time systems. DDS features extensive fine control of real-time QoS parameters, including reliability, bandwidth control, delivery deadlines, and resource limits.

Above, we drew the analogy of publish-subscribe to a newspaper delivery system. That is, of course, an oversimplification. Complex embedded systems have complex data-delivery requirements. DDS is perhaps more like a super television system, with each super-TV set capable of displaying dozens or even thousands of simultaneous channels. Super-TV sets can optionally be broadcast stations; each can publish hundreds of channels from locally mounted cameras to all other interested sets. Any set can add new pictures by subscribing to any channel at any time.

Sets publish many channels, and may add new outgoing channels at any time. Each communications channel, indeed each publisher-subscriber pair, can agree on reliability, bandwidth, and history-storage parameters, so the pictures may update at different rates and record outgoing streams to accommodate new subscribers.

These super-TV sets can also join or leave the network, intentionally or not, at any time. If and when they leave or fail, backup TV set-transmitters will take over their picture streams so no channels ever go blank.

How would you like that in your living room? We hope this gives you some idea of the enormity of the embedded communications challenge. It also outlines the power of publish-subscribe: as you will see, DDS provides simple parameters to permit all these scenarios with a remarkably simple and intuitive model.

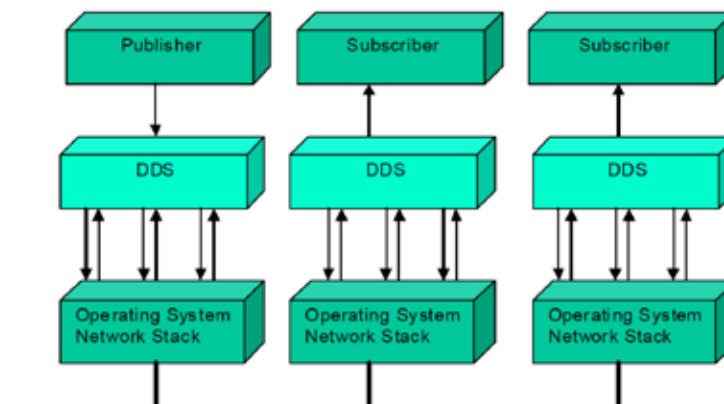While a detailed description of DDS is beyond this paper, we present a brief summary below.

Figure 2. DDS performs the network tasks for the application

Real-Time Publish Subscribe

DDS is fundamentally designed to address the challenges of real-time communications. Sending data at the right time is natural; publishers send data immediately and subscribers get it right away. Data flows directly from source to sink without requiring intermediate servers. DDS supports a "discovery", an intent-registration process that allows nodes or topics to join and leave at any time. Discovery also provides an opportunity to specify per-data-stream Quality of Service (QoS) requirements such as reliability, deadline, and resource usage. Multiple source and sink support allows redundancy and fault tolerance. DDS middleware delivers the right data to the right place at the right time, while handling startups, shutdowns, errors, and specific communication needs.

DDS Model

The DDS publish-subscribe (PS) model connects anonymous information producers (publishers) with information consumers (subscribers). The overall distributed application is composed of processes called "participants," each running in a separate address space, possibly on different computers. A participant may simultaneously publish and subscribe to typed data-streams identified by names called "Topics." The interface allows publishers and subscribers to present type-safe API interfaces to the application.

DDS defines a communications relationship between publishers and subscribers. The communications are decoupled in space (nodes can be anywhere), time (nodes and topics can start, join, or leave in any order at any time), and flow (delivery may be "best effort", reliable, or at controlled bandwidth).

DDS QoS specifies communication parameters between participants, properties of the overall model and of the topics themselves.

To increase scalability, topics may contain multiple independent data channels identified by "keys." This allows nodes to subscribe to many, possibly thousands, of similar data streams with a single subscription. When the data arrives, the middleware can sort it by the key and deliver it for efficient processing.

DDS also provides a "state propagation" model. This model allows nodes to treat DDS-provided data structures like distributed shared-memory objects, with local caches efficiently updated only when the underlying data changes. There are facilities to ensure coherent and ordered state updates.

DDS is fundamentally designed to work over unreliable transports, such as UDP or wireless networks. No facilities require central servers or special nodes. Efficient, direct, peer-to-peer communications, or even multicasting, can implement every part of the model.

Quality of Service

Fine control over real-time Quality of Service (QoS) is perhaps the most important feature of DDS. Each publisher-subscriber pair can establish independent quality of service (QoS) agreements. Thus, DDS designs can support extremely complex, flexible data flow requirements.

QoS parameters control virtually every aspect of the DDS model and the underlying communications mechanisms. Many QoS parameters are implemented as "contracts" between publishers and subscribers; publishers offer, and subscribers request, levels of service. The middleware is responsible for determining if the offer can satisfy the request, thereby establishing the communication or indicating an incompatibility error. Ensuring that participants meet the level-of-service contracts guarantees predictable operation.

Let's look at some examples of important QoS parameters.

Periodic publishers can indicate the speed at which they can publish by offering guaranteed update deadlines. By setting a deadline, a compliant publisher promises to send a new update at a minimum rate. Subscribers may then request data at that or any slower rate.

Publishers may offer levels of reliability, parameterized by the number of past issues they can store to retry transmissions. Subscribers may then request differing levels of reliable delivery, ranging from fast-but-unreliable "best effort" to highly reliable in-order delivery. This provides per-data-stream reliability control.

The middleware can automatically arbitrate between multiple publishers of the same topic with a parameter called "strength." Subscribers receive from the strongest active publisher. This provides automatic failover; if a strong publisher fails, all subscribers immediately receive updates from the backup (weaker) publisher.

Publishers can declare "durability," a parameter that determines how long previously published data is saved. Late-joining subscribers to durable publications can then be updated with past values.

Other QoS parameters control when the middleware detects nodes that have failed, suggest latency budgets, set delivery order, attach user data, prioritize messages, set resource utilization limits, partition the system into namespaces, and more. The DDS QoS facilities offer unprecedented flexible communications control. The DDS specification contains the details.

Is DDS for You?

Complex distributed systems have many tough requirements. There are many factors to consider when selecting distribution middleware. This section outlines driving factors that indicate when applications will succeed with the DDS model.

Complex Data Flows

DDS excels at handling complex dataflow. The finely controlled QoS parameters allow superb integration of systems with modules that require differing update rates, reliability, and bandwidth control. These controls are available on a per-node, or per-stream basis—real advantage when building complex systems.

For example, a ship control system that includes high-bandwidth data sources and sinks, slowly updated graphical user interfaces (GUIs), and long-term logging facilities will make good use of the DDS QoS parameters. While the control system is updated at high rates, the GUI can subscribe at a reduced rate to display the latest state of the system. The logger receives every update reliably, although perhaps with greater latency, to save a complete record of the system operation.

DDS also handles varied transports well, from sporadic wireless connections to high-performance switched fabrics. Systems that include some nodes with fast connections, some with slower connections, and even some with intermittent (e.g. wireless) connections will find DDS provides natural facilities to manage the resulting uneven delivery characteristics.

Need for Speed

Network performance is hard to define. There are many performance parameters, including latency, throughput, failure detection, scalability, start-up time, and repetitive sending rates. While QoS control allows designers to tailor performance aspects to the application, a middleware design cannot simultaneously optimize every performance aspect.

DDS delivers maximal performance to networks comprised of many nodes sending time-critical data. That implies that several aspects of performance are particularly well served by DDS. These include:

- **Tight latency requirements**

  Latency is the time between message generation and receipt. DDS does not require a central server, so implementations can use direct peer-to-peer, event-driven transfer. This provides the shortest possible delivery latency.

  This is a significant advantage over client-server designs. Central servers impact latency in many ways. At a minimum, they add an intermediate network "hop," nominally doubling the minimum peer-to-peer latency. If the server is loaded, that hop can add very significant latency. Client-server designs do not handle inter-client transfers well; latency is especially poor if clients must "poll" for changes on the server.

  In addition, the DDS standard allows applications to trade off reliability for shorter latency. Applications that need the latest data as quickly as possible can further reduce latency by using "best effort" communications.

  The DDS standard also allows applications to take advantage of transports that support prioritization, ensuring even lower latency for the "most important" messages.

- **Large networks**

  Since DDS does not assume a reliable underlying transport, it can easily take advantage of multicasting. With multicasting, a single network packet can be sent simultaneously to many nodes, greatly increasing throughput in large networks. Most client-server designs, by contrast, cannot handle a client sending to multiple potential servers simultaneously. In large networks, multicasting greatly increases throughput and reduces latency.

- **High data rates**

    DDS implements a direct peer-to-peer connection. In addition, the connections are determined at subscription time; when data is ready, every node already knows where to send it. Thus, the actual sending process is efficient, allowing DDS to publish repetitive data at high rates.

Fault Tolerance

DDS is naturally resistant to many of the most common sources of fault.

- **No single point of failure**

    DDS requires no "special" nodes, so it can be implemented with no single-points-of-failure that affect more than the actual lost nodes. Redundancy of publishers and subscribers is built in. Transparent failover is a natural part of the model. Failover can be configured on a per-data-stream basis to satisfy a wide range of application scenarios.

- **Self-healing communication**

    When properly implemented, DDS networks are self-healing. If the network is severed into two halves, each half will continue to work with the available nodes. If the network is repaired, the network will quickly rediscover the new nodes, and once again function as a whole.

- **Support for custom fault-tolerance**

    Implementations are free to add further fault tolerance as well. RTI Data Distribution Service, for instance, supports multiple network interface cards (NICs) on every node. Thus, completely separate networks can be constructed. Even if one network fails completely, the system will continue operation.

Dynamic Configuration

The DDS anonymous publish-subscribe model provides fast location transparency. That makes it well suited for systems with dynamic configuration changes. It quickly discovers new nodes, new participants on those nodes, and new data topics between participants. The system cleanly flushes old or failed nodes and data flows as well.

Application Examples

In this section, we analyze several real-world applications. For each, we examine the specific communication requirements and explore their prime reasons for choosing a publish-subscribe design.

Figure 3. IPC's IQMX VoIP Turret

Voice over IP

IPC builds Voice over IP financial trading terminals with DDS.

Project Description

The IPC IQMX™ VoIP Turret in Figure 4 is designed for the financial market. It lets a stock trader communicate via voice with other stock traders over existing data networks. In addition to one-on-one communications, the IQMX Turret also provides the ability to simultaneously participate in multiple party-line conversations. Data and video communications complement the voice channel. Thus, one platform addresses all the communication needs.

IPC's hierarchical system design has three tiers. The first tier is the primary system center server that communicates with many line cards. The line cards then interface with multiple end-unit IQMX VoIP stations.

A system can have up to 350 line cards, each handling up to 16 IQMX VoIP stations. The system can handle up to 200 calls per second.

Messaging Requirements Filled by DDS

- Dynamic startup and steady state operation: Individual nodes in the environment can start in any order. They can also join or leave at any time. DDS inherently supports these dynamic, ad-hoc networks.

- Efficient database distribution: The publish-subscribe paradigm delivers the data where needed. Direct peer-to-peer messaging makes this efficient.

- Individual message QoS: IPC uses a wide variety of message types. Some messages are cyclic and are sent "best effort" because no confirmation of receipt is required. Other messages must be received by subscribing nodes reliably and in order. DDS reliability control makes this a clean design.

- Heterogeneous communications: IPC uses various operating systems, including Windows®, Solaris™, and VxWorks. DDS provides a consistent application programming interface (API) and transparent conversion between operating system and architecture differences.

Figure 4. Lockheed Martin's Sea SLICE Advance Platform Control System

Sea SLICE, Advance Platform Control System (APCS)

Lockheed Martin has built an advanced ship design called the Sea SLICE (Figure 5). The system uses DDS as the core networking transport.

Project Description

SLICE is an innovative hull and control system design for ships. SLICE limits drag and therefore wake generation. The design also lets smaller ships travel through water conditions that were once considered too harsh to maintain stability.

The Advanced Platform Control System (ACPS) monitors the propulsion system, controls all auxiliary fuel, ballast and diesel generators, and drives the control surfaces of the advanced hull-form dynamics.

Messaging Requirements Filled by DDS

- Reliable messaging platform: SLICE must operate for thousands of hours at sea without failure. SLICE uses DDS's built-in redundancy to enable redundant publishers for critical system-wide commands. If the primary publisher fails, the backup immediately engages to maintain safe system functionality.

- Easy integration for rapid development: The SLICE project was developed under an aggressive schedule. DDS cleanly decoupled the communications design from the functional modules, enabling parallel development teams while easing system integration.

- Multi-platform support: The system uses several different target platforms. A common API and interoperable transport made this simple.

- Test and dynamic configuration: RTI's WaveScope® tool was utilized extensively to view and test the system control parameters. This let Lockheed Martin developers tune their distributed application at runtime—critical for on-the-water performance.

- Dynamic configuration: While the system configuration is static, DDS dynamic configuration was nonetheless useful. During land tests, test platforms could join the network and inject messages to simulate operating and error scenarios. While at sea, test nodes could join and graphically monitor the system operation.

Figure 5. NAVSEA's HiPer-D Project

Navy Open Architecture

The Naval Sea Systems Command (NAVSEA) Dahlgren Division has developed a high-performance radar and ship control system with DDS. This system has evolved into the basis for the Navy Open Architecture standard.

Project Description

NAVSEA Dahlgren investigates advanced technologies for Naval Surface Ship Anti-Air Warfare (AAW). The High Performance Distributed Computing (HiPer-D) project combines radars, user interfaces, and fire control for large systems that encompass multiple ships. The project was originally slated to help Aegis transition from a centralized computer architecture to a COTS-based distributed architecture. It has expanded to a more far-reaching investigation of real-time and distributed technologies for the Navy, and pushing into the other services, now called the Open Architecture (OA) Initiative.

OA targets high-performance sensor, weapons, and integrated control. It has become an important specification feeding into dozens of major programs and new ship designs.

Messaging Requirements Filled by DDS

Complex naval systems require many differing types of communications. DDS provides the capabilities to meet these requirements, including:

- Flexible, data-centric communications: support for one-to-many, many-to-one, and many-to-many data dissemination models provides great flexibility.

- Reliability tuning: Navy designs make particular use of the per-data-stream reliability tuning, allowing the system to handle fast information distribution on a best-effort basis at the same time it handles reliable communications for event notification and alarms.

- Historical data: DDS durability provides configurable data storage on a per-message and per-topic basis. This makes it simpler to introduce new (or recovering) nodes into the system.

- Load-invariant performance: Radar systems must perform reliably when loaded with thousands of tracks. One of the main goals of the Dahlgren work was to demonstrate scalable performance as system demand increased. Dahlgren achieved this goal by adding processors as the load increased. The DDS publish-subscribe data model abstracts data source and sink locations, making process migration much simpler.

- Real-time data performance: Naval systems require efficient distribution of continuously refreshed data, support for high data rate communications, and low latency message delivery.

- Fault tolerance: DDS provides many features that support fault tolerance, such as eliminating single points of failure and providing data-stream "ownership" for automatic failover. Although not required by the DDS specification, implementations such as RTI Data Distribution Service offer redundant network operation and multiple transport support. The result is a highly fault-tolerant system that supports "in-battle" loss of subsystems.

  If any node is lost, the others continue. If an entire network is lost, a second can continue to provide full service. If the network is severed, the two halves continue to operate independently. If reconnected, the split networks will reconnect automatically. DDS provides the basis for a very resilient system.

- Standards-based COTS solution: DDS provides a data-centric publish-subscribe standard, thus promising reduced life-cycle costs, rapid upgrades for technology insertion and refresh cycles, multiple-vendor solutions, development tools, and data analysis tools.
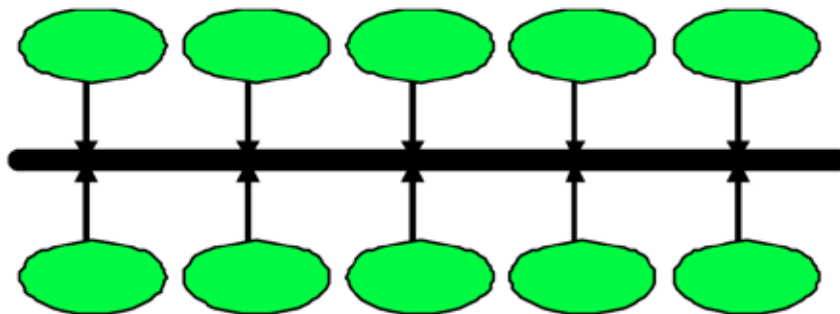

Figure 6. Publish-subscribe decouples data flows

Summary

Publish-subscribe provides a data model that makes complex systems fundamentally simpler to model and understand. If the dataflow diagram above was difficult to draw, try it again with each node just publishing the data it knows and subscribing to what it needs. This design decouples the dataflow.

The best way to draw it is to make a central "data flow bus," and show each node just connected to the bus, as in Figure 7. The data model means you can essentially ignore the complexity of the data flow; each node gets the data it needs from the bus.

If your fundamental problem is data distribution, the right solution is usually obvious. DDS, designed specifically for high-performance data distribution, will greatly simplify your network design. As detailed above, the DDS publish-subscribe model also provides high performance, fault tolerance, fine QoS control, multicast when you need it, dynamic configuration, and connectivity with many transports and operating systems. If you need any or all of these things, you should strongly consider DDS.

About Real-Time Innovations

RTI is the world leader in fast, scalable communications software that addresses the challenges of building and integrating real-time operational systems. RTI Connext solutions meet the needs of enterprise-wide integration—from the operational edge to the enterprise data center. The RTI standards-based software infrastructure improves the efficiency of operational systems while facilitating better decisions, actions and outcomes for the business enterprise.

For over ten years, RTI has delivered industry-leading products and solutions for customers in markets ranging from Aerospace & Defense, Process Automation, Financial Services, Energy, Automotive, Health Sciences and Transportation Management.

Founded in 1991, RTI is privately held and headquartered in Sunnyvale, California.

References

[1] Data Distribution Service (OMG document formal/2007-01-01)

[2] S. Schneider, Build-Your-Own Middleware Analysis Guide

[3] G. Pardo-Castellote and S. Schneider, "Network Data Delivery Service: Real-Time Connectivity for Distributed Control Applications" IEEE International Conference on Robotics and Automation, San Diego, CA 1994

[4] G. Pardo-Castellote, S. Schneider, and R.H. Cannon, Jr., "System Design and Interfaces for an Intelligent Manufacturing Workcell" IEEE International Conference on Robotics and Automation, Nagoya, Japan 1995

[5] Integrated Combat Management System (ICMS), Northrop Grumman Electronic Systems. Document # DS-326-GLV-0404

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