Is DDS for You?

A Whitepaper by Real-Time Innovations

Stan Schneider

Bert Farabaugh

Abstract

Today's embedded software applications are increasingly distributed; they communicate data between many computing nodes in a networked system. Several network middleware designs have arisen to meet the resulting communications need, including client-server, message passing, and publish-subscribe architectures.

The new Object Management Group (OMG) Data Distribution Service (DDS) standard is the first comprehensive specification available for "publish-subscribe" data-centric designs. 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 networking solution.

Introduction

There's a world of opportunity for distributed embedded and real-time applications. The list of applications goes on and on: military systems, telecommunications, factory automation, traffic control, financial trading, medical imaging, building automation, consumer electronics, and more. Each of these industries struggles with how to create a single system from many distributed parts.

The information is out there. The network is fast, cheap, and reliable. We could build whole classes of new distributed applications, if we could just *get the data*. However, getting the data is not always simple. In an embedded network, the main challenge is to find and disseminate information quickly to many nodes. The application must find the right data, know where to send it, and deliver it to the right place at the right time.

All modern operating systems provide a basic network TCP/IP stack. The stack provides fundamental access to the network hardware and low-level routing and connection management.

However, writing directly to the stack results in unstructured code at best. Complex distributed applications require a more powerful communications model.

Types of Middleware

Several types of 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 last decade. Client-server networks include servers, machines that store data, and clients, machines that request data. Most client-server middleware designs present an Application Programmer Interface (API) that strives to make the remote node

©2005 Real-Time Innovations. All Rights Reserved

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. Successful client-server middleware designs include CORBA®, DCOM, HTTP, and Enterprise JavaBeans™ (EJB).

Client-server is fundamentally a many-to-one design. Client-server works well for systems with centralized information, such as databases, transaction processing systems, and central file servers. However, if multiple nodes generate information, client-server architectures require that all the information be sent to the server for later redistribution to the clients. Client-to-client communication is inefficient. The central server also adds an unknown delay (and therefore indeterminism) to the system, because the receiving client does not know when it has a message waiting.

Client-server middleware technologies typically build on top of TCP. TCP offers reliable delivery, but little control over delivery semantics. 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 therefore takes time to set up and significant resources to maintain, 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, such as QNX® and OSE®, use message passing as a fundamental low-level synchronization mechanism. Others provide it as a library service (VxWorks®, Nucleus®, POSIX® message queues). Message-based OS designs can use "send-receive-reply" blocking sequences for inter-node (and inter-process) synchronization and communication. In addition to the message-based operating systems, many enterprise middleware designs implement a message-passing architecture. BEA's MessageQ® and IBM's MQSeries® are significant players in this market.

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's 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 quality of service (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 architectures map well to the embedded communications challenge. Finding the right data is trivial; nodes just declare their interest once and the system delivers it. Sending the data at the right time is also natural; publishers send data when the data is available.

©2005 Real-Time Innovations. All Rights Reserved

Publish-subscribe can be efficient because the data flows directly from source to sink without requiring intermediate servers. Multiple sources and sinks are easily defined within the model, making redundancy and fault tolerance natural. Finally, the intent declaration process provides an opportunity to specify per-data-stream Quality of Service (QoS), requirements. Properly implemented, publish-subscribe middleware delivers the right data to the right place at the right time.

Publish-subscribe designs are not new. Custom, in-house publish-subscribe layers abound. Industrial automation "fieldbus" networks have used publish-subscribe designs for decades. Commercial publish-subscribe enterprise solutions (Tibco's Rendezvous™, JMS) routinely deliver information such as financial data from stock exchanges. In the embedded space, commercial middleware products, including RTI's NDDS®, control complete Navy ships, large traffic grids, flight simulators, military systems, and thousands of other real-world applications. The technology is proven and reliable.

What *is* new is that standards are evolving. The Object Management Group (OMG), the standards body responsible for technologies such as CORBA® and UML®, recently recognized the importance of publish-subscribe architectures. The newly adopted OMG DDS standard is the first open international standard directly addressing publish-subscribe middleware for embedded systems. DDS features extensive fine control of QoS parameters, including reliability, bandwidth control, delivery deadlines, and resource limits. Below, we will take a closer look at DDS and its technical objectives.

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, "send that there" semantics, map 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.

What is DDS?

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 picture-in-picture-in-picture 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.

Each of these sets can also be outfitted with cameras and act as a transmitting station. 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.

©2005 Real-Time Innovations. All Rights Reserved

Figure 1 Super-TV sends and receives many channels

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 (see [1, 3]), we present a brief summary below.

DDS Origins

DDS is fundamentally designed as a real-time embedded technology. It is based on two major product lines: NDDS from RTI and Splice from Thales. NDDS was first developed for a robotic system at Stanford University [4, 5] that has been adapted to over 100 applications, ranging from ship control to traffic monitoring. This paper describes some representative applications. Splice originated as a radar control middleware, also deployed on many real systems [6]. The DDS specification combined the lessons learned by these real-world products with input from academic researchers and customer requirements for new target applications.

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 (delivery may be immediately after publication or later), and flow (delivery may be reliably made at controlled bandwidth).

©2005 Real-Time Innovations. All Rights Reserved

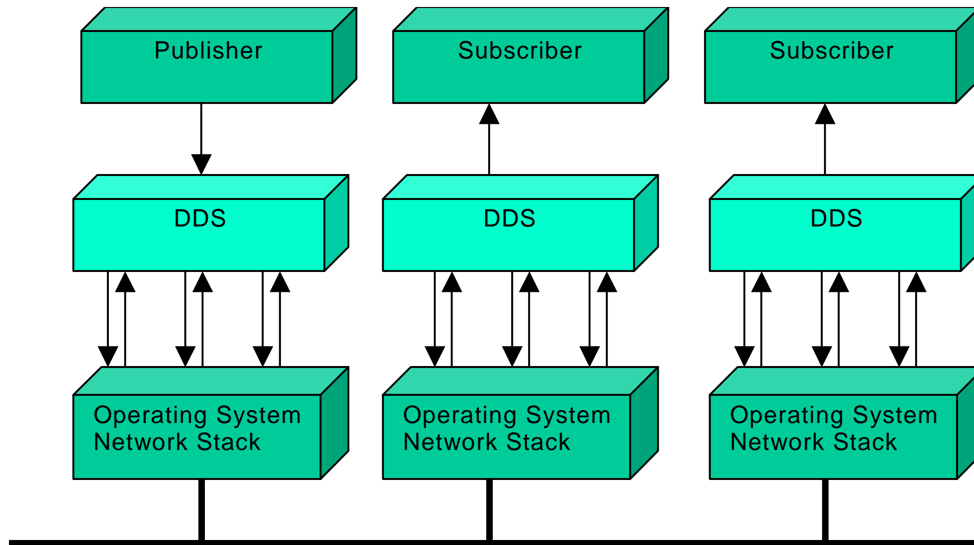

Figure 2 DDS performs the network tasks for the application

DDS QoS parameters specify the degree of coupling 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 structures, with values efficiently updated only when they change. There are facilities to ensure coherent and ordered state updates. An optional part of the specification called the Data Local Reconstruction Layer (DLRL) abstracts most of the underlying publish-subscribe semantics and presents a very-near analog to distributed shared memory.

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 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.

©2005 Real-Time Innovations. All Rights Reserved

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 [1] contains the details.

When is DDS the Right Solution?

As seen in Figure 3, a square peg *will* fit into a round hole if the hole is big enough. When the requirements get tight, however, it's important to have the right peg. Likewise, any middleware will solve some problems. When the requirements are tight, however, it is crucial to select the right middleware.

What does DDS do Poorly?

DDS is fundamentally a data-centric distribution technology. It does not satisfy all communication requirements.

In particular, DDS is not well suited to request-reply services, file transfer, and transaction processing. The data-distribution paradigm is not optimized for sending a request and waiting for a reply, although some implementations leverage the DDS transport to provide this functionality. File transfer, often a one-time request for information, is likely best done with a client-server or TCP-based protocol. Likewise, carefully-controlled sequenced communications, such as reliable, once-and-only once transaction processing, is best provided through other middleware designs.

Therefore, many complex systems require both data distribution and other networked information transfer capabilities. These systems may require combining multiple technologies. Some commercial implementations also build client-server services on top of the data-distribution transport to provide some of this functionality. However, integration of communication models is beyond the DDS standard's (and this paper's) reach.

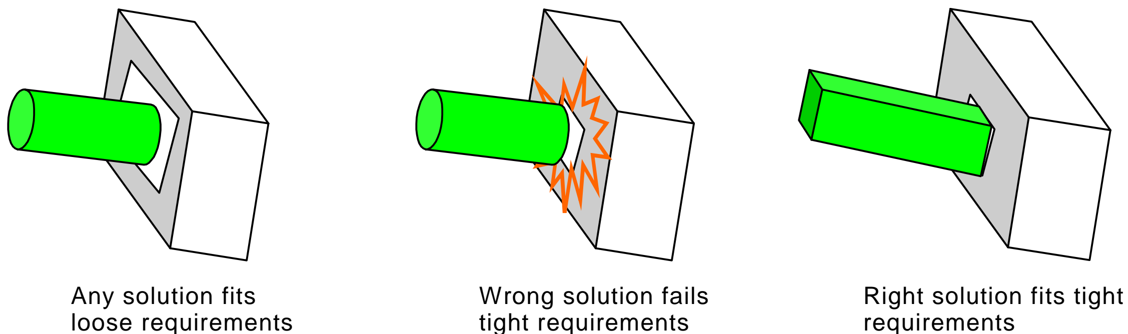

Figure 3 Fitting Requirements to the Right Solution

©2005 Real-Time Innovations. All Rights Reserved

What does DDS do Well?

DDS does many things well. Of course, complex distributed systems have many tough requirements. Matching the peg to the hole is often not a simple process; there are many factors to consider. 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.

©2005 Real-Time Innovations. All Rights Reserved

- *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. NDDS, 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.

Can You Distribute Data in Other Ways?

The data distribution problem is not new. There are of course other ways to distribute data on a network. This section contrasts DDS with some of the other approaches.

The CORBA Notification Service

The Object Management Group (OMG) publishes standards for both the CORBA client-server design and the DDS data distribution paradigm. Both OMG standards address middleware. However, they target different needs, as documented above. Nonetheless, CORBA does specify an event-driven service known as the notification service (NS). This facility offers a "data push" model of communications in contrast to the client-server based CORBA model. The NS differs from the Data Centric Publish Subscribe (DCPS) model of DDS in several ways.

The NS is event-centric and provides data only as a description of the event. DCPS is data-centric and provides an event only to notify of changes in the data.

Thus, in NS, "channels" function as event streams. Data appears piggybacked to each individual event. Consumers select the events of interest by binding to the appropriate channel and installing filters. The intent is to have few channels and have consumers select the events of interest by installing suitable filters.

In DDS, each data stream functions as a channel. Definition, configuration, and QoS properties are all specified relative to each data stream. The data itself is structured and this can be used to provide further refinement in the data channels, such as via the "key" facility. Filtering within a stream is limited and based on either timing or contents of the data. The net effect is to create many "channels," each representing a data stream.

DDS implements a clear data model. Each named topic is treated as a unique entity. Thus, for instance, repeated updates of a topic could be

©2005 Real-Time Innovations. All Rights Reserved

ignored, or even thrown out intentionally, by the middleware, since they are known to be related. Events in NS, by contrast, are independent entities that happen to have attached data.

The target application domain is also different. DDS focuses on efficient real-time data-delivery. It is designed to scale to many suppliers and potentially thousands of recipients. It stresses flexibility by accommodating fine QoS control for reliability, timeliness, and robustness as described above.

The NS, by contrast, targets distributed event notification through a few known channels. Recipients must monitor these channels and filter out the data of interest. It targets occasional or sporadic notification to a few recipients.

For a detailed analysis, see the discussion on the notification service in the DDS RFP [7].

Reflective Memory and Switched Fabrics

Reflective memory and switched fabrics are hardware solutions to the distributed data problem. With a reflective memory system such as SysTran's SCRAMNet®, each node has a special board that contains some memory and a high-speed connection (usually fiber) to other boards in the system. When data is written to one board, the hardware quickly delivers it to all the other boards in the system.

Hardware solutions can deliver higher performance than software-based solutions. However, that performance comes at the price of expensive hardware, restricted range, and few options for redundant connections or allowance for failed nodes. It also doesn't implement a data model that manages and organizes the data flow.

DDS implementations can use shared memory or switched fabrics as transports, thereby combining top performance with a clean data model.

Roll Your Own

Writing your own custom middleware is often a viable alternative. Custom middleware has the advantage of being a better fit to the problem. It usually has the disadvantages of a risky schedule, unclear or changing requirements, difficulty in anticipating future needs, lack of tool support, cost of maintenance and support, and…it's a lot of work. Still, there are often compelling reasons to make the build-it-yourself decision. The issues are beyond this paper; see the "Build-Your-Own Middleware Analysis Guide" [2] for a much more complete analysis.

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.

Note: As of this writing, the DDS standard is just being finalized, so there are no implementations. All the highlighted applications are using NDDS, RTI's middleware product that preceded and strongly influenced the DDS standard. Note also that there are hundreds of applications, ranging from industrial automation to highway traffic monitoring and control. These are a somewhat representative set; more are available at www.rti.com.

Unmanned Underwater Vehicles

Bluefin Robotics produces various Unmanned Underwater Vehicles (UUVs) with DDS technology.

Project Description

The Bluefin vehicles perform autonomous ocean surveillance in areas that are too dangerous for manned missions. Typical payloads include side-scan radar, synthetic aperture sonar and hydrophone arrays. Other sensors including Doppler Velocity Logs, Acoustic Doppler Current Profilers, Conductivity and Temperature, Fluorometer, and GPS assure mission reliability. For each mission, the UUV is preprogrammed onboard the host ship and then deployed. The UUV returns to its origination point after the mission, and mission data is then retrieved and analyzed.

©2005 Real-Time Innovations. All Rights Reserved

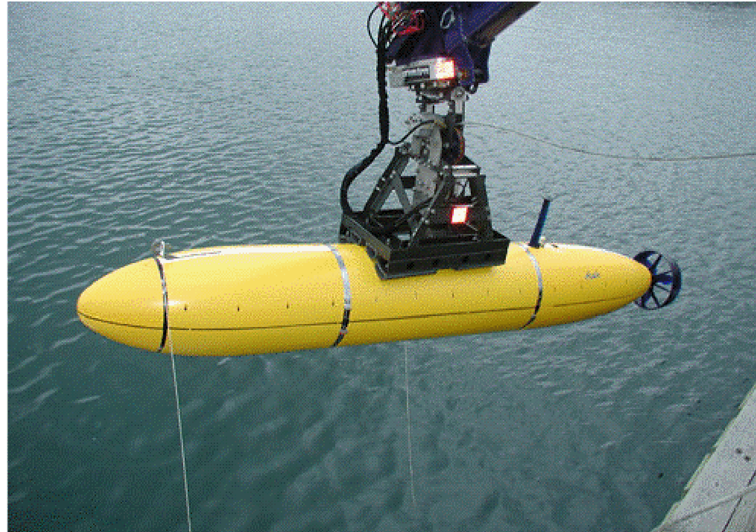

Figure 4 Bluefin Robotic's UUV

DDS provides the primary communication between the Main Vehicle computer, the Payload computer, and the Operator computer. There are approximately 20 processes on the main vehicle computer using DDS via a high-performance shared-memory transport.

Messaging Requirements Filled by DDS

- Low latency: The streaming sensor data have time-critical information required by the main vehicle computer. Best-effort communications for all sensor data provides Bluefin with a low latency communication.
- Per-message QoS: DDS gives Bluefin developers the ability to specify individual communication parameters on a per-data-stream basis. Thus, each communication path can have its own reliability, determinism, flow control, and redundancy specifications.
- Modularity: DDS completely decouples the communications problem from the system architecture. The resulting system is implemented cleanly as functional modules with a simple interface to a network "data bus."
- Heterogeneous environment: DDS transparently connects the system's various operating system and processor combinations. Bandwidth and QoS control make it possible for different modules to coexist on the same bus.
- Future expandability: abstract publishers and subscribers allow Bluefin to migrate existing processes to multiple vehicle computers without having to redesign the communication platform.

Voice over IP

IPC builds Voice over IP financial trading terminals with DDS.

Project Description

The IPC IQMX™ VoIP Turret in Figure 5 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.

©2005 Real-Time Innovations. All Rights Reserved

Figure 5 IPC's IQMX VoIP Turret

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.

Sea SLICE, Advance Platform Control System (APCS)

Lockheed Martin has built an advanced ship design called the Sea SLICE (Figure 6). 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.

©2005 Real-Time Innovations. All Rights Reserved

Figure 6 Lockheed Martin's Sea SLICE Advance Platform Control System

- 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.

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

©2005 Real-Time Innovations. All Rights Reserved

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

Figure 7 NAVSEA's HiPer-D Project

©2005 Real-Time Innovations. All Rights Reserved

eliminating single points of failure and providing data-stream "ownership" for automatic failover. Although not required by the DDS specification, implementations such as RTI's NDDS 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.

So, Is DDS for You?

Is DDS for you? That, of course, depends on your needs. For most applications, the primary driver is to find the lowest-risk path to high-performance data distribution.

Although the alphabet soup of network protocols may seem confusing, choices for building embedded middleware come down to only a few options: roll your own, a client-server design such as DCOM or CORBA, a message-based OS or message-passing middleware, and DDS.

Roll Your Own, if…

It makes sense to write your own middleware if:

- you consider it one of your core competences, or
- your needs are not met by any middleware, or
- your needs are (very) simple, and unlikely to grow.

Many organizations consider building their own middleware because the full-blown products seem like "overkill." Don't be put off by extra features. Any professional middleware (or software for that matter) has features you don't need. That doesn't mean it won't satisfy your needs. It doesn't make sense to rewrite Microsoft Word because you don't need an online thesaurus. Besides, your needs may change; you may never know how helpful a thesaurus is until you have one online. The important point: determine if the middleware does what you need. Unless they handicap you, don't be deterred by features you don't think you'll use.

Even if you have traditionally chosen to write and maintain your own networking layer, it pays to track the middleware products. Middleware is becoming vastly more functional. Standards are greatly lowering the risk and driving competition. The day will come, as it did for real-time operating systems, when the only ones who write their own are those with very special needs or those with very special budgets.

Use Client-Server (DCOM or CORBA), if…

Use client-server if your system dataflow fits the basic design. How do you know? Diagram your system's dataflow. This doesn't require a detailed design; a quick map of information sources and sinks usually suffices. Focus on which nodes will have the information, how they will find each other, and how the data will flow.

©2005 Real-Time Innovations. All Rights Reserved

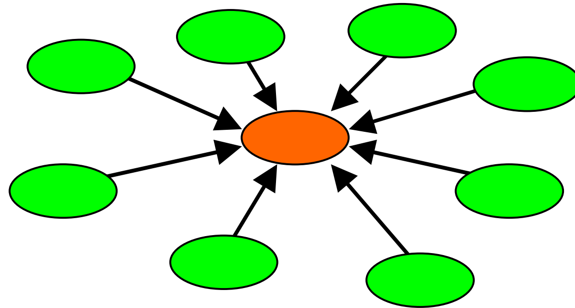

Figure 9 Client-server works best with centralized data

If your drawing looks like a "hub-and-spoke" system (see Figure 9), a web server, or a centralized database, then client-server will work well for your application. In most systems that are well suited to client-server architecture, it is easy to specify where the servers should be. The relatively static information sources are centralized. Clients rarely need to talk to other clients, and if they do, the communications are not time critical.

Several other characteristics indicate that a client-server design will work best. Usually, most (all) transactions are easily modeled by "request-reply" semantics. Replies are often large, e.g. big files. Processing proceeds as a series of steps. Time criticality and fault tolerance are second-order issues.

If you have a hub-and-spoke architecture with these properties, select DCOM if your system is restricted to nodes using the Windows operating system, CORBA otherwise. Also consider other client-server transports, such as HTTP.

Use Message Passing, if…

A message-passing design is best if you don't need a data model. How do you know if you need a data model? Your design drawn above may not fit the hub-and-spoke, but it will have a definite simple structure. Successful message-passing designs usually look like plumbing supply lines into a neighborhood, as seen in Figure 10. A few main information trunks deliver data, which may branch out to several destinations. Most flow is one-way and relatively static. Return lines may or may not follow roughly the same patterns.

Since the sources and sinks of any particular data item are well known from the beginning, it's not important that the middleware help the application figure out where to send things. The application sets up the queues and then uses them to send information. The sending node therefore knows where the data is going ahead of time.

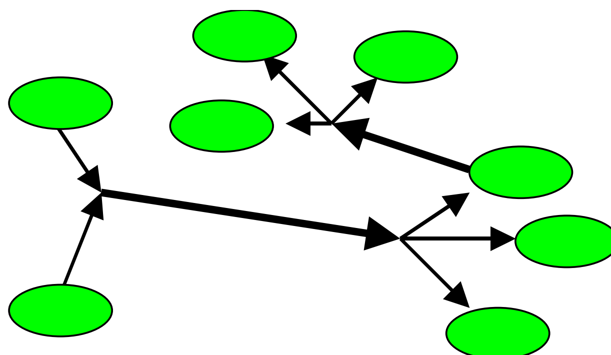

Figure 10 Message passing works best with a few clear channels

©2005 Real-Time Innovations. All Rights Reserved

Be a little careful: if you see nodes that want to "tap into" the plumbing to get data, that's an indication that publish-subscribe may be more appropriate.

Use DDS, if…

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 11. 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.

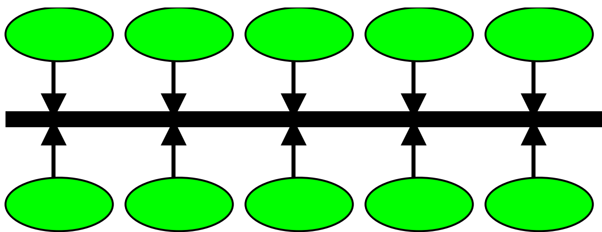

Figure 11 Publish-subscribe decouples data flows

References

[1] Data Distribution Service (OMG document ptc/2004-03-07)

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

[3] G. Pardo-Castellote, B. Farabaugh, and R. Warren, "An Introduction to DDS and Data-Centric Communications"

[4] 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

[5] 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

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

[7] Data Distribution Service for Real-Time Systems Request For Proposal (OMG document orbos/01-09-11.pdf)

Real-Time Innovations, NDDS, RTI, and WaveScope are registered trademarks of Real-Time Innovations, Inc. All other names mentioned are trademarks, registered trademarks, or service marks of their respective companies or organizations.

©2005 Real-Time Innovations. All Rights Reserved