Using Switched Fabrics and Data Distribution Service to Develop High Performance Distributed Data-Critical Systems

Dr. Rajive Joshi
Real-Time Innovations, Inc.

High performance and predictability are prerequisites for any large-scale networked system dependent on real-time data processing and analysis. Data representing actual events or system status must be evaluated while it is still relevant to tactical conditions, making it imperative to know when specific data is available to aggregate and evaluate that data in real time. Unreliable receipt times make effective analysis difficult or impossible.

Fast and predictable performance is always an issue in the design of a large-scale networked system dependent on real-time data processing and analysis, but especially so when designing distributed systems with thousands of nodes that need to move a lot of data around quickly in a dynamically changing environment. Switched-fabric networks [1, 2] can provide fast and highly scalable hardware solutions and are now being increasingly used in such applications. What is needed beyond that is a software solution for bringing predictability, flexibility, and reliability to distributed data communications. I describe how the Data Distribution Service (DDS) [3] data-centric publish-subscribe middleware layer can realize the full potential of a hardware switched fabric network to deliver a complete solution for application developers.

Data-Critical Systems Share Characteristics

Many large-scale, data-critical applications can be characterized by three attributes: the need to gather and distribute data in real-time, the large amount of data being transferred, and the entities involved in this data exchange are varied and may even change over time. For instance, air traffic control, financial transaction processing, battlefield, naval command and control, or industrial automation systems all are examples of data-critical systems which have these three attributes.

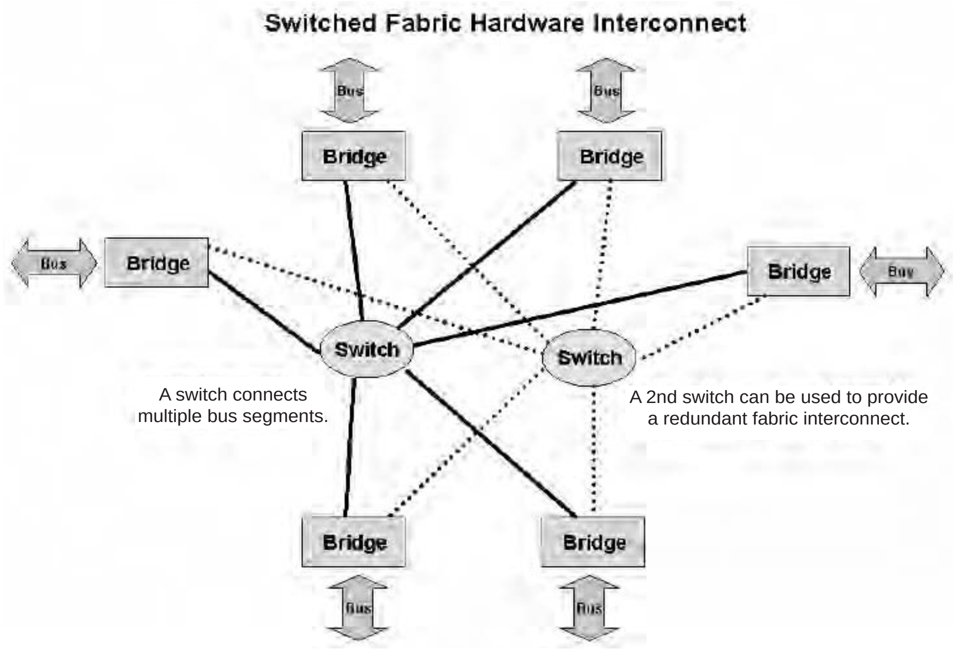
These systems are not necessarily hard real-time, but their predictability requirements are an integral part of the functions they perform. They gather data from a variety of sources, sensors for example, and they distribute the data to a variety of users like databases, display devices, or control algorithms. Furthermore, by their very nature, they are distributed.

Today's bus-based architectures, typically multi-Central Processing Units (CPU), Versa Module Europa (VME) backplane solution with hard-wired input/output (I/O) interfaces to sensors and effectors, fall short in several areas in addressing the needs of data-critical systems. For example, these hardware transport mechanisms do not scale, are difficult to make fault-tolerant, and are difficult to modify and upgrade once they have been deployed.

For these reasons, designers of complex, data-critical distributed systems are turning to switched fabrics to replace bus backplane and serial interconnect technologies. StarFabric, Peripheral Component Interconnect (PCI) Express Advanced Switching, Serial Rapid I/O and InfiniBand are some commercial products that implement different switched fabric designs [1, 2].

A switched-fabric bus is unique in that it allows all nodes on a bus to logically interconnect with all other nodes on the bus (Figure 1). Each node is physically connected to one or more switches. Switches may be connected to each other. This topology results in a redundant network or fabric, in which there may be one or more redundant physical paths between any two nodes. A node may be logically connected to any other node via the switch(es). A logical path is temporary and can be reconfigured, or switched among the available physical connections. Switched fabric networks can be used to provide fault tolerance and scalability without unpredictable degradation of performance, among other features.

Figure 1: *Switched Fabric Architecture. Multiple Switches Can Be Used to Expand the Fabric and Provide Hardware Redundancy*


Switched Fabric Hardware Interconnect
Bus
Bridge
Bus
Bridge
Bus
Bridge
Bridge
Bus
Switch
Switch
A switch connects multiple bus segments.
A 2nd switch can be used to provide a redundant fabric interconnect.
Bridge
Bridge
Bus
Bus

Switched Fabrics and Data Distribution Service

A key characteristic of switched fabrics is that they allow peer-to-peer communication between nodes without having to physically connect every node to every other node. With every node physically connected to every other node, adding a new node is exponentially more and more expensive as the number of nodes increases. Because a switched fabric network employs switching to achieve logical connectivity and reconfigurability, these systems can be architected to be highly scalable.

On the software side, publish-subscribe communication systems naturally map onto switched fabrics. Publish-subscribe systems work by using endpoint nodes that communicate with each other by sending (publishing) data and receiving (subscribing) data anonymously via topics. A topic is identified by a name and a data type. A data producer declares the intent to publish data on a topic; a data consumer registers its interest in receiving data published on a topic. The middleware acts as the glue between the producers and the consumers; it delivers the data published on a topic by a producer to the consumers subscribing to that topic. There can be as many topics as needed – a producer can publish on multiple topics and a consumer can subscribe to multiple topics. The middleware layer isolates the data producers from the consumers; they have no knowledge of each other.

A publish-subscribe software architecture allows producers and consumers to be loosely coupled. As a result, it is naturally scalable and can easily adapt to the changing needs of distributed data-critical systems. The producers and consumers are peers – they directly communicate with each other, so the topology of publish-subscribe systems can be closely matched to that of switched fabric systems. Thus, a publish-subscribe middleware layer can fully exploit the potential switched fabric network hardware.

DDS standard (see The DDS Standard sidebar on page 28) specifies a data-centric publish subscribe middleware layer, developed with the needs of distributed data-critical applications in mind. A well-designed DDS middleware implementation can be good at real-time data distribution, be easily field-upgradeable, and be transport agnostic. It can be better at real-time data distribution because publish-subscribe is more efficient than the traditional request/reply based architectures in both latency and bandwidth for periodic data exchange. Further, applications can be easier to upgrade in the field because publishers and subscribers do not care who or how many their counterparts are. And finally, since the middleware is layered on top of the physical means of getting the data from one place to another, it does not need to depend on the network transport or topology used.

Figure 2 illustrates the DDS data-centric publish-subscribe architecture. A topic has a name and a data type associated with it and represents the application data model. DataReaders and DataWriters are associated with topics. A DataWriter can publish data on its associated topic; a DataReader can subscribe to data on its associated topic. DDS middleware automatically and anonymously sets up direct data flows between DataWriters and DataReaders associated with a topic, resulting in scalable and fault-tolerant data distribution.

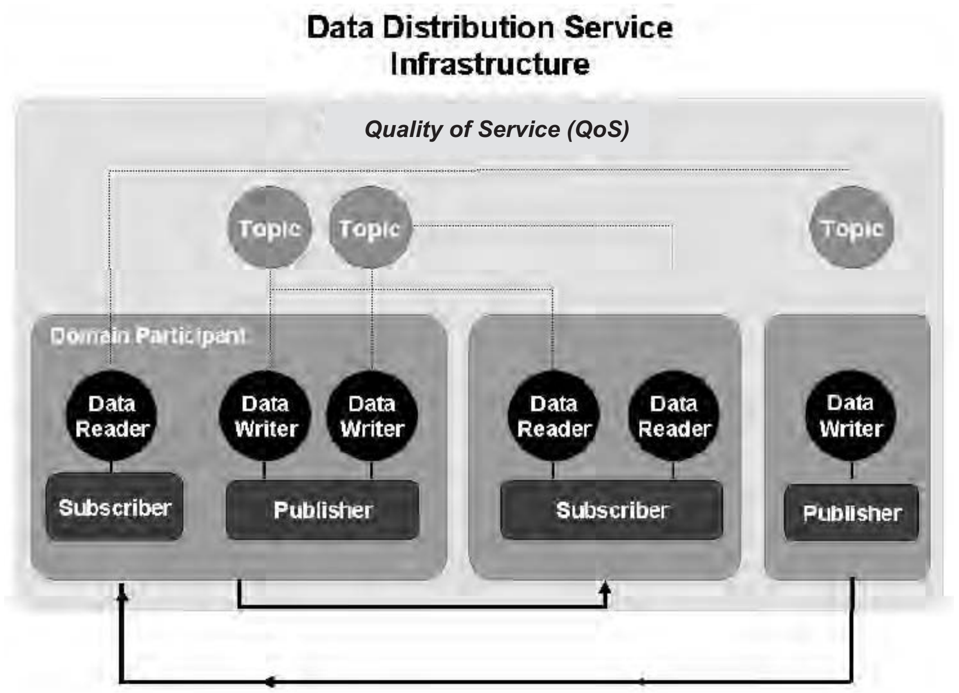


Figure 2: *DDS Data-Centric Publish-Subscribe Architecture Organizes the Data in a Distributed System Around Topics*

New Choices for System Architects

This marriage of switched fabrics and DDS real-time middleware offers architects new flexibility in adding capabilities that were once much more difficult to achieve. Many of the features offered by switched fabrics have complementary capabilities in the DDS-compliant middleware. For example, switched fabrics typically offer rich error management features such as the ability to recognize, report, and route around failed paths. With DDS-compliant software, system designers can also take advantage of DDS error reporting facilities.

A key feature of switched fabrics is support for multiple paths between nodes. This gives system architects the ability to easily implement multiple physical interconnects that can be combined with sophisticated error management. Likewise, with DDS, applications can take advantage of redundant publishers that have different strengths. When a higher strength publisher fails, a lower strength one is automatically switched in by the DDS middleware. In addition to fault tolerance, this can also help with load balancing on heavily used networks.

Switched fabric specifications already provide for a hot plug or hot swap capability. This hardware capability can be combined with a *virtual* hot plug capability at the application level using DDS middleware. Unlike traditional tightly coupled client/server architectures, DDS middleware allows producers and consumers to be dynamically added or removed in an operational system.

Many switched fabrics provide sophisticated features that allow, for instance, bandwidth-reserved, isochronous transactions across the fabric, something that is not supported by, say, Ethernet. Corresponding to the hardware QoS facilities, DDS-compliant middleware can offer a number of QoS policies that make predictability at the application level possible. For instance, the TRANSPORT_PRIORITY policy allows developers to manage how they prioritize one data flow over another.

The Road Map for Distributed Data Services

The existence of DDS as a standard specification endorsed by the Department of Defense (DoD) paves the way for addressing the challenge of distributing data among a myriad of defense systems. DDS is now mandated for data distribution by

The DDS Standard

The DDS for Real-Time Systems standard [3], from the Object Management Group (OMG), defines a publish-subscribe system that has high performance, is efficient, and offers a predictable way of meeting the data distribution requirements of data-critical systems with minimal overhead. The standard can be found on the OMG web site at <www.omg.org/technology/documents/formal/data_distribution.htm>.

The DDS standard has been in existence for almost two years, maps very naturally to the topologies and capabilities of switched fabrics, and is maturing into a solid technical approach to managing data distribution across large-scale distributed networks.

The DDS standard has three main goals:

1. To define a model for communication as pure data-centric exchanges, where applications publish (supply or stream) data which is then available to remote applications that are interested in it.
2. To provide a mechanism of specifying the available resources and providing policies that allow the middleware to align the resources to the most critical requirements, giving system designers the ability to control Quality of Service (QoS) properties that affect predictability, overhead, and resource utilization.
3. To permit systems to scale to hundreds or thousands of publishers and subscribers in a robust manner.

the Navy Open Architecture Computer Environment [4] and DoD Information Technology Standards Registry [5] and has already been adopted by programs such as Future Combat Systems, DD(X), Littoral Combat Ship, and Ship Self-Defense System.

But, despite the existence of a standard specification, the value of the solution is highly dependent upon its implementation. The specification defines certain features and capabilities, but not how they should be implemented.

A carefully designed middleware architecture can reduce the likelihood of a fault, limit the damage of a fault if it does occur, help detect faults immediately, protect the middleware from errors in application code, and isolate applications from errors in other applications. That architecture can also deliver significant advantages in the performance and flexibility of network distributed data communications.

For example, the DDS specification defines how a publish-subscribe communication model should work for a distributed real-time network. The DDS specification defines DataWriters for publishing and DataReaders for subscribing to a single topic on a user-defined data type. This in itself is standard and straightforward but how this is implemented can have a significant impact on network performance and scalability.

A robust implementation improves both performance and scalability by defining an architecture that supplies each DataWriter or DataReader with a queue that buffers messages bound for another endpoint through a transport. This architecture supports direct end-to-end messaging, since each endpoint (a DataReader or DataWriter) in each application directly communicates with a sister set of endpoints. Each endpoint has a dedicated set of buffers to hold messages in transit to other endpoints. This queuing architecture provides for an optimized transfer of messages from DataWriter to DataReader, no matter where each resides on the network. And because the endpoints queue and buffer transmissions to other endpoints, this architecture can easily scale to large and complex networks still with predictable delivery times.

In a similar manner, DDS defines the concept of a DomainParticipant, which is the fundamental container entity that can participate in a publish-subscribe network. A DomainParticipant can contain many DataReaders and DataWriters. Typical applications may use only one domain, and therefore have one DomainParticipant. However, applications are free to create several DomainParticipants so multiple instances of this entity can exist simultaneously.

Multiple execution threads are a way to optimize responsiveness and performance while also allowing the system to scale across a broad fabric-based network. One possible approach is to use several dedicated threads for each DomainParticipant in the following manner:

- **Event Thread.** DDS allows application designers to associate various QoS policies with each topic and data flow between a DataWriter and DataReader. These include timing related QoS that are implemented by the middleware. The Event thread manages both timing delays and periodic events such as protocol heartbeats, deadlines, and liveliness needed to meet the QoS policies requested by the application.
- **Database Cleanup Thread.** This thread purges old information from the internal data structures such as publication declarations and subscription requests.
- **Receive Threads.** A port represents a transport specific resource for receiving incoming messages. Data packets are delivered to transport's ports. Different DataReaders can be configured to receive messages on different ports. In order to minimize the end-to-end latency, a receive thread is created per port provided by the transport.

When the application writes new data to a topic, the message passes all the way through the middleware down to transport level send in the caller's thread. In the user's thread context, the message is serialized, deposited into the writer queue, encapsulated into a wire-protocol packet, and passed to the transport for delivery.

In the common case, the entire operation's critical path takes no inter-application locks and suffers no context switches. The event thread is only involved if the initial transport operation fails, or to execute follow-on processing such as ensuring reliable delivery. The event thread has ready access to the message since it is already stored in the writer queue. When the transport receives a new packet, the appropriate receive thread processes the packet, retrieves the message, stores it in the reader queue, and immediately executes the listener callback. In the common-case critical path, there are no inter-application locks or context switches. If the application requires the message to be handled with user threads, it can do so with DDS WaitSets. Both flexibility and performance are optimized, even as the network scales.

Performance can also be impacted through the poor use of the code execution path. Since lock contention can have a significant detrimental impact on performance, fast path optimization takes data to or from the network transport to the application using a single lock per message, greatly simplifying the resource sharing protocol.

Finally, instead of using lists to store the information needed to dispatch and manipulate messages, hash tables can be used. Although hash tables are more complex than lists, they have constant time access provided that the initial allocation of space is sufficient. Regardless, in the worst case, access time is logarithmic, which is better than linear linked lists.

The Implementation Optimizes Performance, Flexibility, and Reliability

Performance, flexibility, and reliability represent just a few ways that an implementation of the DDS specification can impact the three critical characteristics of data communication over a distributed network – reliability, performance, and flexibility. Alternatively, a poor implementation of the DDS specification can mean that the architecture works well under certain optimal implementations, but fails to take advantage of greater resources, and fails to scale as the network grows.

Data communications system developers do not want to change their application code when the fabric is updated, changed, or augmented. However, many possible implementations can deliver suboptimal results when the network topology changes. A DDS implementation can take this into account so that the application can be easily re-optimized to deliver a comparable level of performance in the face of evolving and changing fabrics.

As switched fabric technology advances, the middleware must support those advances by being able to adapt to new transport mechanisms and different resource requirements and availability. Being able to plug-in different transports in the middleware layer makes it possible to more easily incorporate new fabric technologies as they become available without making any changes at the application layer.

A superior implementation of the DDS standard enables network performance to be optimized to the particular application. It matches the performance needs with the underlying fabrics and availability of system resources such as memory. The design's flexibility allows it to target a broad array of applications and network topologies by supporting many transports and maintaining individual resources for each connection. Finally, the design avoids most key single points of failure, increasing reliability.◆

References

1. Arshad, Nauman, Stewart Dewar, and Ian Stalker. "Serial Switched Fabrics Enable New Military System Architectures." COTS Journal Dec. 2005 <www.cotsjournalonline.com/home/article.php?id=100438>.
2. Cotton, David B. "Switched Fabrics and the Military Market." COTS Journal. (Apr. 2005) <www.cotsjournalonline.com/home/article.php?id=100294>.
3. OMG. "Data Distribution Service for Real-Time Systems, v1.1." Document formal/2005-12-04. (Dec. 2005) <www.omg.org/cgi-bin/doc?formal/05-12 -04>.
4. Dahlgren Laboratory. Navy Open Architecture Computing Environment <www.nswc.navy.mil/wwwDL/B/OACE/>.
5. Defense Systems Information Agency. DoD Information Technology Standards Registry <https://disronline.disa.mil/>.

About the Author

Rajive Joshi, Ph.D., is a principal software engineer at Real-Time Innovations, Inc., where he specializes in the design of distributed and real-time systems. He has served as a consultant and developer for distributed systems projects in the areas of robotics and automation, including Alstom Schilling's Quest's remote-operated vehicle and CrouseHinds' automated filament-alignment system. Joshi is the author of *Multisensor Fusion: A Minimal Representation Framework*, and is a member of the Institute of Electrical and Electronics Engineers, Association for Computing Machinery, and American Institute of Aeronautics and Astronautics, Inc. He holds a doctorate and a Master of Science degree in computer and systems engineering from Rensselaer Polytechnic Institute, and has a Bachelor of Technology in electrical engineering from the Indian Institute of Technology in Kanpur, India.

Real-Time Innovations, Inc.
3975 Freedom CIR 6th FL
Santa Clara, CA 95054
Phone: (408) 200-4700 ext. 4754
Fax: (408) 200-4702
E-mail: rajive.joshi@rti.com

Letter to the Editor

Dear CROSSTALK Editor,

As always, I thoroughly enjoy CROSSTALK articles and most everything that comes out of the Software Engineering Institute.

I am of the considered opinion, after more than 42 years of development experience, that the problems with software quality can be attributed to a single cause, that being the inability to recognize complexity and act accordingly.

Resulting in: Do we tackle problems beyond our capability to solve using human intellect, excluding mathematical processes, even at the module level (unknown high levels of McCabe's Cyclomatic Complexity Index)?

Does anyone other than the Cleanroom Software Engineering sequence enumeration requirements analysis process folks identify and count the number of state transitions in a single module, much less a whole process or system? Or understand the implication of having 64 bimodal variables that can occur in any combination and various legal sequences? Or understand that programming is the mapping of state transitions to code?

Does anyone understand that the implication of Brooks' Law is a loss of intellectual control over the process and product as the process and staff and product grows in size and complexity?

Is complexity a self-inflicted wound? For example, could the failed IRS project been designed into multiple cases? Could a separate system have been created to process the majority of filers (e.g. 1040EZ), rather than a monolithic, all cases design? I expect so and would have been a quick success, though I recognize we may still be developing the more complex cases when Gabriel blows his horn. Yoder calls such monolithic design A Big Ball of Mud <www.laputan.org/mud/>.

Do programmers exacerbate the problem by not being able to defend their own or their team's performance, such as 12 defects per thousand lines of code for a medium complexity module?

Thanks for listening!

– Carl Wayne Hardeman
<cwhardeman@yahoo.com>