# Design strategies for real-time data in distributed systems

**by Mike Rogosin,** Real-Time Innovations

THIS ARTICLE DESCRIBES HOW DATA CAN BE EXCHANGED IN DISTRIBUTED SYSTEMS. DDS, DESIGNED SPECIFICALLY FOR HIGH-PERFORMANCE DATA DISTRIBUTION, WILL SIMPLIFY A COMPLEX NETWORK DESIGN. DDS IS NOW PUBLISHED BY OMG AS AN OPEN INDUSTRY STANDARD, PROVIDING A UNIVERSAL PLATFORM FOR SYSTEM INTERCONNECT.

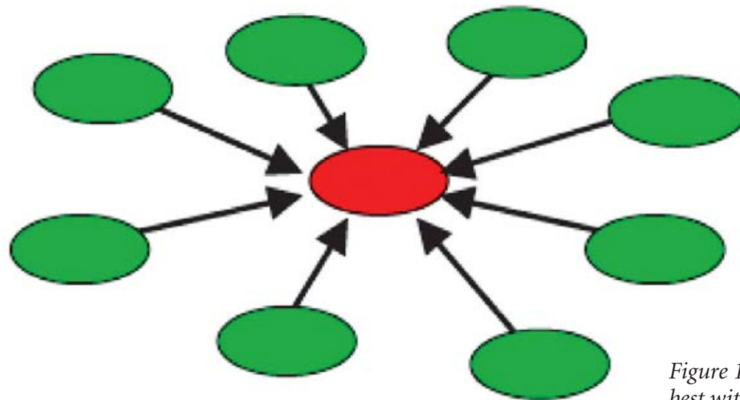

*Figure 1. Client-server works best with centralized data*

Today's embedded software applications are increasingly distributed. A wide range of standards-based COTS products are available allowing the configuration of cheap and reliable systems which communicate real-time data between many computing nodes at high speed. So why is the design of larger, more complex distributed applications still such a challenge? The main issue is that efficient handling of real-time data is not always as simple as it looks. An embedded network must find and disseminate information quickly to many nodes. The application needs to find the right data, know where to send it, and ensure delivery to the right place at the right time. This is of course not a new problem.

Virtually 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. Several types of software technologies, commonly known as "middleware", 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" 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 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 are also generating information, client-server architectures require that all the information be sent to the server for subsequent redistribution to the clients. Such indirect client-to-client communication is inefficient, particularly in a real-time environment. The central server also adds an unknown delay to (and therefore removes determinism from) the system, because the receiving client does not know when - or even if - 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 takes time to set up and significant resources to maintain, TCP does not scale well for extended data distribution in larger systems.

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 (e.g. 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

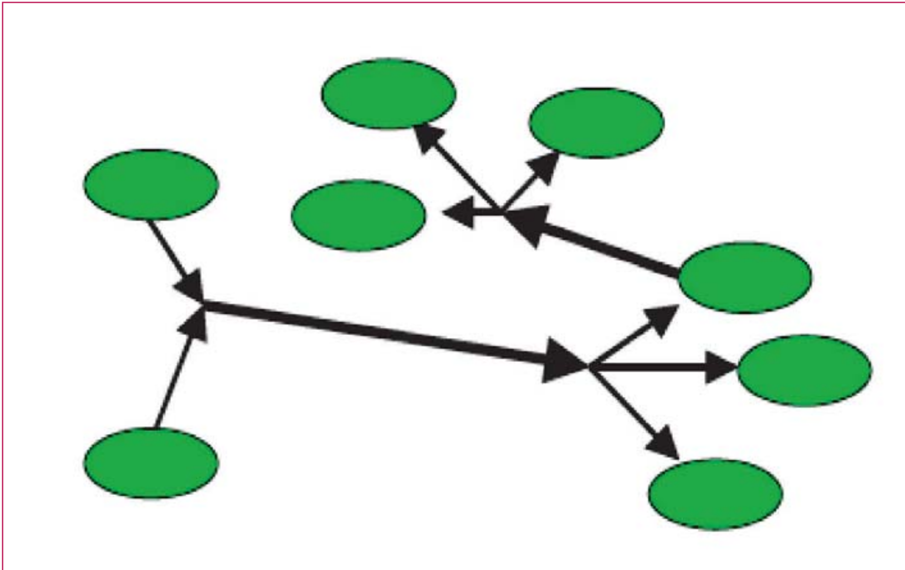

*Figure 2. Message passing works best with a few clear channels*

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 the application knows where that process/channel is, what happens if that channel doesn't exist, etc. The application developer 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. Lastly, in the embedded space at least, it usually creates a dependency on a particular OS being present throughout the system, raising issues of application portability and integration with nodes outside the OS.

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 provides 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. 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, modern implementations of publish/subscribe middleware, such as data distribution services (DDS from the OMG) allow elegant and efficient quality of service (QoS) mechanisms to be specified per data stream. Properly implemented, publish-subscribe middleware delivers the right data to the right place at the right time. Of course, publish-subscribe designs are not new. Custom, in-house publish-subscribe layers abound. Industrial automation "Fieldbus" networks have used simple and mostly hardware-dependent 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 naval ships, large traffic grids, flight simulators, military systems, and thousands of other real-world applications. The technology is proven and reliable.

What is new here is that capabilities are improving and 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 ad-

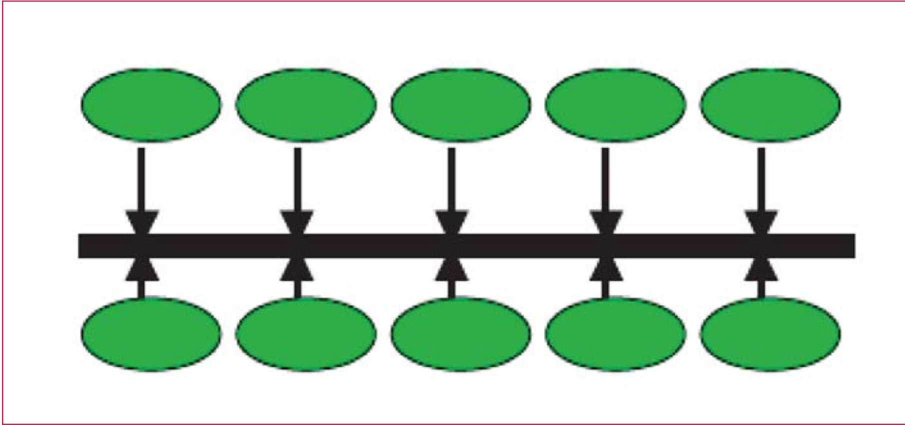

*Figure 3. Publish-subscribe decouples data flows*

dressing publish-subscribe middleware for embedded systems. Also, DDS's advanced features include extensive fine control of QoS parameters, ensuing defined levels of reliability, bandwidth control, delivery deadlines, and resource limits.

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 inherently provides both discovery and messaging services. 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.

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. If your drawing looks like a "hub-and-spoke" system (see figure 1), 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 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.

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 2. 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. 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.

Is publish-subscribe right for your application? If the primary driver is to find the lowest-risk path to high performance real-time data distribution, and where system scalability and an open standards-based environment may be important additional factors, then the answer is probably yes. 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 3. The data model means you can essentially ignore the complexity of the data flow; each node gets the data it needs from the bus.

DDS, designed specifically for high-performance data distribution, will simplify a complex 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. Best of all, DDS is now published by OMG as an open industry standard for data distribution, thus providing a universal platform for system interconnect. If your applications need any or all of these capabilities, you should take a closer look at DDS. ■

**Click-for-More**

Interested in more information about design issues and interface standards in real-time distributed embedded systems?

Visit our specific website with links to:

- A resources page with information on the DDS and Corba standards, as well as technical articles, design notes and white papers
- A detailed analysis by the University of Helsinki of the performance of RT Publish-Subscribe models in embedded systems.
- A technical summary of the issues involved in using DDS and the IPv4 to IPv6 Transition
- A Technical Seminar on DDS and Distributed Data Management given by RTI, TimesTen and 4Tec.

Simply type-in Reader Service #: 624 at **Embedded-Control-Europe.com/know-how**