What Real-Time Data Distribution System Is Right for You?

A wide range of standards-based COTS products are now available for the configuration of today's increasingly distributed embedded software applications, 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 not a new problem. Virtually all-modern operating systems provide a basic network TCP/IP stack. While the stack provides fundamental access to the network hardware and low-level routing and connection management, writing directly to the stack results in unstructured code. Complex distributed applications often require a more powerful communications model.

Several types of software technologies, commonly known as middleware, have emerged to meet the needs of these complex distributed applications. They fall into three broad classes: client-server, message-passing, and publish-subscribe.

Client-Server

Client-server networks include machines that request data (clients) and machines that store data (servers). Most of these middleware designs present an Application Programmer Interface (API) that strives to make the remote node appear to be local. That is, users call methods on remote objects in order to get data, just as if they were on the local machine (also called Remote Method Invocation, or RMI). Client-server designs work well for systems with centralized information, such as databases, transaction processing systems, and central file servers. Successful client-server middleware designs include CORBA®, DCOM, HTTP, and Enterprise JavaBeans™ (EJB).

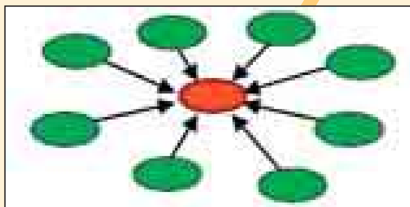


Figure 1: Client-server works best with centralized data.

How do you know when client-server fits a system's dataflow? A quick and simple diagram mapping information sources and sinks will usually suffice. It should focus on which nodes will have the information, how they will find each other, and how the data will flow. If the drawing is many-to-one, meaning that it looks like a "hub-and-spoke" system (see Figure 1), a Web server, or a centralized database, then client-server will work well for the application.

If multiple nodes are generating information, client-server architectures will require that all 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 if it has a message waiting.

In most systems that are well suited to client-server architecture, it is easy to specify where the servers should be because 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 also indicate that a client-server design will work best:

- 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 when your system is restricted to nodes using the Windows operating system. Select CORBA otherwise. Also consider other client-server transports, such as HTTP.

Keep in mind that client-server middleware technologies typically build on top of TCP (Transmission Control Protocol). TCP offers reliable delivery, but little control over delivery semantics. For instance, TCP retries dropped packets even though the retries may take a lot of time. TCP also requires dedicated resources for each connection and does not scale well for extended data distribution in larger systems due to the set-up time and maintenance needs of each connection.

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, making it easier to exchange information between many nodes in the system. Some operating systems (e.g., QNX® and OSE®) use message passing as a fundamental low-level synchronization mechanism; others provide it as a library service (e.g., VxWorks®, Nucleus®, and POSIX® message queues).

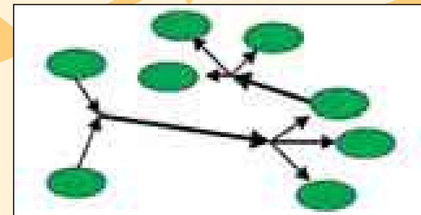


Figure 2: Message passing works best with a few clear channels.

Message-based operating system (OS) designs can use send-receive-reply blocking sequences for inter-node (and inter-process) synchronization and communication. In addition to the message-based OS, 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.

A message-passing design is best if you do not need a data-centric model: With this architecture there is no real model of the data itself, only a model of a means to transfer data. How do you know if you need a data-centric model? The system design drawing will have a definite simple structure, even though it may not fit the hub-and-spoke pattern.

Successful message-passing designs usually look like plumbing supply lines into a neighborhood (see Figure 2). A few main information trunks deliver

Intro Cov ToC

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. If you see nodes that want to tap into the plumbing to get data, publish-subscribe may be more appropriate.

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 does not address how the application knows where that channel is, what happens if that channel does not exist, etc. The application developer must determine where to get and send data, and when to do the transaction.

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. In the embedded space, 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

Publish-subscribe adds a data model to messaging. To find the right data, nodes declare their interest once and publishers send data when it is available. Messages pass directly between the communicating nodes (source to sink) without requiring intermediate servers. Multiple sources and sinks are defined within the model for natural redundancy and fault tolerance. The fundamental communications model provides both discovery (what data should be sent) and delivery (when and where to send it). These systems are good at distributing large quantities of time-critical information quickly, even in the presence of unreliable delivery mechanisms.

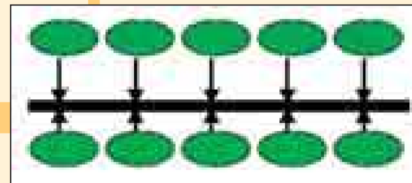


Figure 3: Publish-subscribe decouples data flows.

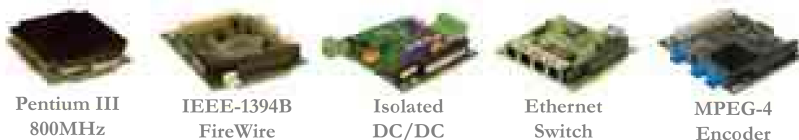
Modern implementations of publish-subscribe middleware allow QoS mechanisms to be specified per data stream. Properly implemented, publish-subscribe middleware delivers the right data to the right place at the right time. Publish-subscribe capabilities continue to improve and standards are evolving. The Object Management Group's (OMG, a standards body responsible for technologies such as CORBA and UML®) newly adopted Data Distribution Services (DDS) standard is the first open international standard directly addressing publish-subscribe middleware for embedded systems.

Is publish-subscribe right for your application? If the message-passing 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 (see 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.

This article was written by Dr. Robert Kindel, vice president of Technical Services for Real-Time Innovations (RTI), located in Santa Clara, CA. Contact Dr. Kindel at bob.kindel@rti.com or (781) 306-0470, ext. 202, for more information.

Engineering Rugged COTS Systems with Rugged PC/104 Modules

New Fanless -40°C to +85°C Rated Products:



With a full arsenal of rugged embedded PC solutions coupled with seasoned experience building defense-rated systems for land, air and sea, Parvus is an expert at rugged COTS systems. Our boards, chassis and systems are made to reliably perform under high stress conditions: extreme temperatures, shock/vibration, humidity, and high altitude. See our solutions onboard the F/A-22, EA-6B, F-16, HMMWV, M48, EFV and many other platforms. Contact Parvus today to see how we can help you with your next project.

www.parvus.com | 877-223-1557

For Free Info Visit http://info.ims.ca/5215-861