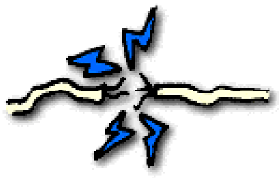
Message Queuing Middleware: No longer high risk for I/S

Though implementing the technology can be very complex and expensive, using the right technology at the right time can do wonders for a business



By Max Dolgicer

Middleware is an essential component for developing any large scale distributed applications -- such development requires real application architecture and application integration. There are two fundamentally different approaches to application architecture and application integration. In the first approach, databases are the center of the universe. In the second, applications exchange information in real or near realtime.

This story examines the now rapid evolution from the first method to the second. For years, I/S organizations saw message oriented middleware, or message queuing, as a high risk solution. Today that risk is diminishing quickly.

For more than two decades, companies have mostly used the first approach to application integration -- with mixed success. In its basic form this approach integrates applications either by sharing or exchanging databases. Until recently, every business in the universe spent substantial money and kept many happy (or maybe not so happy) programmers on the payroll to support what is fundamentally called application integration. Shared databases were a mandatory component of such integration.

The profound changes occurring in today’s environment are driving application integration to new dimensions. A key requirement is an ability to exchange information at realtime or near realtime. Growing a business requires enhanced customer service, which requires a single, near realtime view of each customer. The global view that many businesses are taking forces more business applications to be interrelated, which in turn forces applications to exchange information as never before.

Today, simple business transactions are rarely performed in a vacuum. A typical transaction is usually complex, multifaceted and affects a number of applications. While successful data exchange creates new business opportunities, it also averts disasters. Finally, the web is a catalyst that accelerates the need for information exchange.

The bottom line is that applications integration drives the need for reliable and timely information exchanges. The scope of such information exchange can either be a single application where multiple processes need to exchange information, or multiple applications exchanging information within the same or different business domains, or even across separate enterprises.

Ultimately, enterprises look for easy to use, easy to develop, and easy to manage software that can enable applications to exchange information regardless of underlying hardware platforms, operating systems, and of target applications. In short, companies are looking for application "glue" -- middleware -- that can provide a reliable way to exchange information. Today there are multiple middleware categories that enable process-to-process or application-to-application communication. This article focuses on message oriented middleware, or message queuing middleware, one of the fastest growing middleware categories.

This article is the first of two parts. This installment describes the components that comprise a typical message queuing system, and describes the functionality such systems should perform. Next month's installment will compare and contrast IBM's MQSeries, the undisputed market leader, and upstart contender MSMQ, just shipped by Microsoft as part of the Windows NT 4.0 service pack.

To be fair, there are other very mature message queuing products on the market, and even a number of very mature message passing products. Nonetheless, at the moment, MQSeries is the undisputed message queuing king that Microsoft is aiming to displace.

It is not too late to successfully compete with MQSeries if a contender can offer either: a superior product; multiple broad distribution channels; the Microsoft aura. Thus the decision to compare MSMQ and MQSeries. Microsoft definitely qualifies in the last two conditions. And MSMQ is an industrial strength product that in some areas is superior to MQSeries.

Message Queuing Middleware is no standard. There is no common source code, as there is with DCE, and no common set of specifications as is the situation with the Corba object request broker standard set by the Object Management Group (OMG), Framingham, Mass. Further, "The Essential Corba" treatise on middleware by Tom Mobray and Ron Zahavi, dedicated only half a page to message oriented middleware, and the pair dismissed the technology as immature.

Last year, OMG issued a Request for Information (RFI) for asynchronous messaging within Corba, which would extend the ORB's synchronous communication model to an asynchronous one. Observers ask what asynchronous messaging adds to Corba that isn't available through the Corba Event service. We can shows that by using the Corba Event service, Corba client objects are decoupled from directly communicating to Corba server side objects. This question will be answered in much more detail in future stories. For now, this story will just note that the OMG RFI focuses on Quality of Service (QoS), which allows applications to add properties such as guaranteed and assured delivery to application requests and replies. Corba Event Service does not deal with QoS altogether. OMG is in the business of defining standards for interfaces, not how these standards are implemented. Therefore, the debate over how to implement asynchronous messaging will continue after an OMG standard is adopted.

Despite the lack of any formal standards, message queuing is already a mature technology. Hundreds of mission critical applications in some of the largest corporations are running in production using message queuing. These applications are spread across every industry and are implemented using IBM's MQSeries and several other solutions available today. For example, hundreds of mission-critical applications running at Swedish automaker Volvo uses internally development message queuing software dubbed VCOM. VCOM is now marketed by Veri-Q Inc., San Francisco. Simply put, when VCOM is down, Volvo cannot manufacture cars. Fortunately, VCOM is very stable and Volvo production does not stop. The same type of stability and performance is applicable to a number of other message queuing products including MessageQ from BEA Systems Inc., Sunnyvale, Calif., NetWeave DS from NetWeave, Smartsockets from Talarian Corp., Mountain View, Calif. and X-IPC from Momentum's Software Corp., Hasbrouck Heights, N.J.

The common vocabulary and requirements for applications built using message queuing products include support for 7 x 24 x 365 operations, support for hundreds of concurrent users, and an ability for graceful recovery in case of failure.

A message is simply strings of bits and bytes that have meaning to one or more application programs. Often business requests represented by messages contain data, control or both. In the world of message queuing, messages are used for programs within the same application or in different applications to communicate with each other. However, unlike other types of middleware such as DCE where different programs communicate using straight remote procedure calls, message queuing middleware programs do not communicate directly, but rather indirectly using message queues. These messages also can be of different types, including requests, replies to request and exceptions reports.

Message queues are named entities (IBM calls them objects in the most abstract sense of the word) where messages are stored by a given application program, and are retrieved by another application program. Queues can be explicit -- exposed to the application programmer -- or implicit -- not exposed. IBM and Microsoft products both expose the queues within the respective APIs. Message queues also can be of different types and have properties which describe them. Options can include local or remote queues, memory based or disk based, and transactional or non-transactional.

Meanwhile, message queuing is defined as a method of program-to-program communication. The method calls for different programs within a single application or separate applications to communicate by writing and reading messages to and from queues.

These applications have no dedicated and logical connections between them.

This definition raises at least one legitimate question -- how does one application know what message format the other application program expects? The short answer is that it doesn't. Most message queuing products have no knowledge of the necessary message format needed to be defined by two applications in order to enable a meaningful communication. Most message queuing products treat application messages simply as a series of bytes. In other words, message queuing products are unaware of application content. All message queuing API's are dealing with manipulating message queues and facilitating message movements within the network by simply reading and writing messages to and from message queues.

There is no standard Interface Definition Language available as in Corba where message formats supported by application objects are defined. In fact, each message queuing product adds a control header (such as the IBM MQMD -- message descriptor) where control information is kept. In fact, such a header is added to each application message. Generally, a control header contains information such as Queue Names, Queue Properties, indicators for Quality of Service, Message Identification, message type (request, reply, report on exception) and others.

The process of controlling message queues can be confusing for users. Different message queuing products have different designations for the software that is responsible for controlling message queues. For this story, the term, "Queue Manager" is appropriate. The Queue Manager system software provides message queuing services to application programs. A Queue Manager can play multiple roles and handle multiple responsibilities. For example, the software is responsible for queue management including creation and deletion of queues; provides an API that allows application programs to communicate; provides the communication services necessary to assure that messages are moved across the network; can be responsible for routing messages intelligently throughout a network and for dealing with different levels of message confirmations and acknowledgments. Of course, a message routing service could be implemented as a separate product component dedicated to providing routing service to a Queue Manager.

A typical application using message queuing can perform a number of operations under a few assumptions. Examples of such assumptions are: programs communicating indirectly with each other with queues used to store and retrieve messages; programs using the services of Queue Manager to communicate with remote application programs; and applications not concerned with messages being lost in transit. Users can assume that a typical message queuing system guarantees delivery of a message from the source queue to the destination queue, which can be on the same or remote machine relative to the source queues.

For application developers, the message queuing paradigm is very simple, involving two basic operations -- the sending of a message to a queue and the receiving of a message from a queue. Unfortunately the simplicity can be very deceiving -- designing applications using message queuing is not simple. Careful design considerations must be applied when building distributed applications using message queuing.

Nonetheless, very sophisticated application architectures can be designed and implemented using message queuing. Such application architectures can include publish/subscribe, hub and spoke as well as plain client/server. Under the publish/subscribe architecture, applications that publish information and subscribe for information can use the message queuing system to distribute messages reliably and with guaranteed delivery. Using hub and spoke architecture, many applications can post messages reliably to a central hub from where it will be distributed reliably to many other applications, thus enabling many-to-many applications communication. Lastly, message queuing products can be used for enabling client to server communication in client/server architectures.

A typical application flow between programs using a message queuing paradigm is:

- Both programs (sender and receiver) attach to the message queuing engine which is generally 	represented by the Queue Manager.

- The sending program sends a message to a target queue that can be located on the same machine (local queue) or on a remote machine (remote queue). In the remote queue, the message is safely stored locally before it reaches the remote queues.

- The receiving program is started by the arrival of a message in its queue. This step is called triggering. Alternatively, receiving programs can "listen" on the queue while waiting for a message to arrive. Of course, one of the basic principles of message queuing is to accommodate situations when the sender and receiver are not running at the same time. Message queuing successfully deals with situations when the receiving program is not ready to receive the message. The message is stored on the queue until read by the receiving program.

- After processing a message, the receiving application replies to the reply-to queue.

- When the information exchange is complete, both sides are detached from the respective message queuing engine.

While the flow described above could be an oversimplification, it accurately portrays the general flow of events for program-to-program communication using message queuing.

Much of the difference between message queuing products is based on whether or not the queues and other important components of a given message queuing offering (queue managers, network "moving" service) are exposed programatically, administratively or both.

Regardless of the complexity of the API, using message queuing services to design applications does require extra work and considerations. As stated, the design process of applications using message queuing is NOT straightforward like the use of traditional synchronous communication mechanisms such as remote procedure calls or SQL calls (when both the sender and the receiver are time dependent and require both sides of the application and communication resources to be available). It should be noted, that since programs running in the message queuing paradigm do not communicate directly nor maintain a state of information, any link can break. As a result, additional design considerations need to apply. The application design needs to accommodate answers to many questions, such as: what happens on a message queue is full? Or, what happens if an application request expires or could not be delivered to a target application within a specified time? Or, will many application processes read from a single queue or it will be one queue per each application process? The bottom line is that the flexibility offered by message queuing systems in application design is not without cost. However, because of its flexibility (many styles of application architectures can be developed using message queuing) and relative programmatic simplicity, message queuing products are nevertheless very popular. Furthermore, to integrate applications using message queuing products does not require totally different mindset as the situation is with Corba, which requires programmers to learn new object-oriented skills.

Message queuing solutions can offer multiple functions to users, including:

- MULTIPLATFORM AND MULTIPROTOCOL SUPPORT. A message queuing product should support multiple communication protocols and multiple operating systems. After all, message queuing is used to integrate applications in modern enterprises that run heterogeneous operating systems and communication protocols. As long as there are multiple operating systems and platforms, support for heterogeneous operating systems and communication protocols is a requirement.

- QUALITY OF SERVICE (QOS). Quality of Service encompasses many important features. Guaranteed and assured message delivery, and message priorities are just a few of the features which are under the umbrella phrase "Quality of Service." The first feature guarantees that application requests or replies (represented by a message or series of messages) are delivered to the target application regardless of network reliability. This feature is important over most network types, and is vital over intermittent networks. The guaranteed delivery feature keeps application developers from having to deal with network outages, which decreases programming complexity and eliminates unnecessary and costly exception handling.

Guaranteed message delivery is needed to deal with situations where receiving applications are not available to accept a request. For example, a settlement application will not necessarily be ready to accept messages from a trading application. However, the message queuing system should guarantee that when the settlement application is ready to receive messages, the request will be delivered immediately.

Assured delivery is also an important property of Quality of Service. It guarantees that a message is delivered only once. Such a feature is vital in applications that deal with financial transactions. Of course in the business world, timeliness is everything and most applications should exchange information in a timely manner. Therefore a message queuing system should be capable of handling expiration requests, when a request is automatically discarded within a set time period. Of course, the sending application should be notified when a particular request has expired.

- TRACKING AND NOTIFICATION. An application using a message queuing should be able to track requests, and to request notification. It should be able to track when a request is delivered to the remote queue and when the target application receives it. An application should also be able to request notifications when an event occurs such as a request expired or a request received by the target application.

Notification and tracking go hand in hand with the ability of a message queuing system to generate different types of events. Every message queuing system should also have a built-in event instrumentation mechanism so events such as a queue full condition can be logged and interpreted by the system management components of the system. Events should be generated when a message request arrives to queue, which would allow a message queuing system to start an application with the arrival of a message to a queue. Since message queuing promotes time independence where applications do not have to be running at the same time in order to exchange information, the ability to support triggers is an important feature. Since triggering allows an application instance to be started when a message arrives at a queue, an application need only be active when work is required, which avoids unnecessary resource consumption.

- LOCATION TRANSPARENCY. Neither the sending nor receiving application should be concerned with the physical location of message queues. Both applications should only be concerned with queue names. From a developers point of view, there is no difference whether the queue is local or remote.

- SERVICE TRANSPARENCY. Using message queuing applications to exchange messages in order to perform a business transaction or a service. The mapping between the service name and the underlying queue names should be transparent.

- QUEUE ORDERING. Message queues can be ordered using different ordering. FIFO, LIFO, priority or ordering based on a user defined field are examples of how message queues can be ordered.

- GETTING APPLICATION MESSAGES. Programming flexibility in the process of getting messages is important for users. Multiple methods should be provided by the message queuing product. These can include: receiving application waits on a queue until a message arrives at the queue; the receiving application polls the queue until a message (any message or a specific message) arrives at the queue; or the receiving application registers a callback, which in turn is executed when a message arrives at the queue. The first two methods are synchronous and the third is purely asynchronous and event driven.

- TRANSACTION SUPPORT. Ideally transaction support has three requirements: (a) support for internal transactions; for example, allowing an application to send and receive messages within a single transaction coordinated by the message queuing engine; (b) transaction support coordinated with database updates; for example, allowing an application to send (or receive) messages to the queue (or from the queue) and update a database in a single transaction, without requiring a transaction monitor; and (c) support for external transactions; for example, when queuing operations and updates to a database are coordinated by an external transaction monitor. It should be noted that when applications use transactions, all messages are delivered exactly once, and in the order that they were sent.

- MULTIPLE LANGUAGE SUPPORT AND ACTIVEX INTEGRATION. Support for many programming languages, including Java, is important. However, many users complain about the inability of message queuing products to provide an easy integration with desktop applications and development tools. Support for ActiveX is important to alleviate this problem. Support for ActiveX, which can be viewed as a technology bridge between many desktop applications and middleware, makes it easy to access message queuing functionality from the wide range of desktop applications like Excel, application development tools such as PowerBuilder, and programming languages like VisualBasic, Visual J++ and Visual C++.

- THREADS SUPPORTS. Thread support has become a given for any enterprise middleware. Message queuing is not an exception. Threads support is expected in two areas, the API, which must be multi-threaded, and the kernel (the message queuing engine) itself. Thread support at the API level will allow an application to manipulate queues from multiple threads without forcing each thread to open and close queues. Queue handles, generally obtained at the time of open queues operation and referred later when send and receive operations are performed, should be shared across mutiple threads.

- FLEXIBLE IMPLEMENTATION OF A MESSAGE QUEUING ARCHITECTURE. Message queuing should be implemented in two modes -- client or server. Message queuing servers are capable of performing queuing operations and can be ideally configured to perform additional functionality such as intelligent routing. Clients are generally dependent on the servers that connect to perform sending and receiving operations.

Clients are not capable of performing any queuing operations. Some companies prefer to use the client/server mode because it requires less administration for defining components such queues and queue manager that comprise message queuing product. To define queues and queue managers can be a monumental task, especially for large scale distributed applications. The cost of administration, configuration and management of message queuing products can be very expensive.

- ADMINISTRATION, CONFIGURATION AND MANAGEMENT. To configure message queuing products can be complex, thus the software requires an ability to administrate and configure every component from a central location. The ability to manage messages, queues and queue managers is a key requirement. Many observers feel strongly that the management aspects of any message queuing products have to be built into the product. Unfortunately, many of today's products don't offer this feature. For example, the MQSeries requires either a third party product or a home-grown tool to add such functionality. People buying the MQSeries management product quickly discover shocking news -- to buy a MQSeries management product today costs more than an MQSeries itself. Clearly in order to administer, configure and manage message queuing products efficiently, a robust naming service is required.

- NAMING SERVICE. Queue names and locations of other components of message queuing systems need to be stored in a repository such as naming service. Existence of a dynamic naming service should reduce the amount of administration and configuration and also provide the ability to correlate application topology (application names and service names that application provide) to queue names.

The above requirements represent only a partial list of requirements. More features such as intelligent routing, load balancing security and Internet support are just a few additional requirement that come to mind.

Clearly, message queuing technology and its product represent a viable and not high risk middleware solution for applications integration and distributed application development. However, the simplicity of API is very deceiving. IT planners, system architects and developers need to take into consideration that designing asynchronous and event-driven application based on message queuing technology can be complex. To deploy and manage such applications can be even more complex and expensive. In the second part of this series we compare how both, MQSeries and MSMQ are addressing some of our requirements. Stay tuned.

This article appeared in the December 1997 issue of Application Development Trends.