Microsoft's Enterprise Middleware Strategy

by Max Dolgicer

Is it ready for prime-time?

Love it or hate it, Microsoft wants to host your applications and transform itself from the company that sells Microsoft Office upgrades every year into a major provider of enterprise software.

They aspire to be what IBM was-and still is, in the eyes of many corporate decision-makers. Given that Microsoft wants to be everyone's system software provider, their only viable strategy is to offer high-quality solutions across a spectrum of requirements. After making Windows the dominant desktop standard, the next logical step for Microsoft is to become the dominant standard in the enterprise application space. Just a few years ago, though, the spirit of NT was willing, but the enterprise application software was weak. Because the key to serious enterprise application software is scalable and function rich middleware, Microsoft did not offer any middleware besides COM, which worked only on the local machine, at first. Going after market opportunity, however, has never been Microsoft's weakness, and in the last few years, Microsoft has developed substantial enterprise-level application support by shipping a number of serious middleware products, such as distributed versions of COM (i.e. DCOM), MTS, and MSMQ. MTS and MSMQ products are capable of providing enterprise-level application support that enables consistent access to services and applications throughout the enterprise. Of course, Microsoft is not alone in the pursuit of this ever-growing middleware market, in general, and the object middleware market, in particular.

Once an enterprise makes a commitment to use distributed object technology to build distributed, mission-critical applications, it must decide which object middleware to use and which distributed architecture to endorse. Up until now, the traditional logic has been to leave the desktop to COM, and to use products based on OMG?s CORBA standards with its CORBAServices such as transactions, security, and naming as enterprise-level middleware.

Despite all the alternatives, it is now clear that Microsoft entry into the middleware market his changed the market profoundly by challenging the assumption that industrial-strength middleware is solely the providence of a few CORBA, TP monitor, and Message-Oriented Middleware vendors. In fact, Microsoft's challenge to the middleware market is similar to Netscape's challenge to Microsoft's dominance of application presentation front ends.

Is OS-Bundled Middleware a Good Thing?

One of the things that has driven middleware development has been a requirement to provide developers with transparent access to functionality across heterogeneous operating systems and communication protocols. Within reason, the computing environment should be irrelevant. However, in the real world, some middleware products are often optimized for specific platforms, or offer more features on one platform than another. Sometimes this can be the result of corporate history, marketing strategies, and large-scale development opportunities with make-it-or-break-it customers. As a result, middleware product purchasing decisions can sometimes be based on support for a specific target operating system.

More often than not, corporations must choose between a consistent, cross-platform implementation and one that best exploits the services of a specific operating system. As usual functionality and interoperability are at opposite ends of the spectrum here. Tight integration with an operating system usually comes at the expense of cross-platform consistency. It is unrealistic to ignore the issue of specific operating system support.

COM is an integral part of the Windows operating system. Though it is available on operating systems other than Windows from Microsoft as well as its partners (Software AG, for example), Microsoft’s focus continues to be on innovating COM for Windows operating systems. Everything that Microsoft does is COM- and Windows-related. Nobody, not even Microsoft?s biggest adversaries, can blame Microsoft for being inarticulate about their strategy - a strategy which, in our opinion, will continue into the foreseeable future.

Although cross-platform interoperability is a major mission for middleware, the strategy of single-minded middleware benefits Microsoft and Windows-based customers. The strategy of bundling middleware with Windows makes it virtually free, and assures a successful market penetration for COM, as well as for additional middleware products such as Microsoft Transaction Server (MTS) and Microsoft Message Queue (MSMQ).

On the server side, for price-minded customers (and for customers who made a decision to standardize on two operating systems, such as Windows and OS/390), it has not been easy to justify any object middleware besides Microsoft’s own Common Object Model (COM). On the client side, the statistics say it all - the presence of COM signals Microsoft?s continuing dominance. But what about distributed applications? Is COM alone enough? If not, will MTS and MSMQ combine with COM to provide a ready-for-prime-time middle-ware object infrastructure?

What Is COM?

The Component Object Model (COM) implements components as language- Independent encapsulated objects that communicate with other components (and applications) by exposing a standard set of structured interfaces. Distributed COM (DCOM) allows COM components to communicate through-out a network.

In terms of distribution, COM objects can be local or remote, and standalone or in-process (see Fig. 1). Clients can access local COM objects on the same machine, or remote COM objects on a different machine. Standalone processes are executables (.EXE files), while in-process objects are implement-ed within a dynamic link library (DLL) that must execute in some kind of container, because a library cannot run by itself. Although all of these combinations look the same to the client, components perform faster and use fewer resources when they are implemented as DLLs.

There, are some advantages to using COM for component deployment. A COM object can publish its interfaces, which leads to several benefits. This allows objects to add new features to be discovered without risking the integrity of clients that are ignorant of them. It also provides the flexibility of multiple inter-face inheritance. Versions, an indispensable requirement for any component development, are just robust extensions in COM.

Because of its OS-bundling strategy, Microsoft has succeeded in penetrating the market with COM. This makes it easy for vendors to choose COM as a risk-averse platform for both system and application components. This, in turn, provides a wide variety of competitive sources for any enterprise that is looking at a "buy, don’t build" approach to creating application systems.

In fact, COM’s language-independent and binary-compatible standard seems to be designed to allow a true mass market for components. This, coupled with COM’s integration with many development and productivity tools, seems to make the long-awaited world of "applica-tion building blocks" an inevitable reality. However, there have been a few stumbling blocks thus far.

Ignoring the market realities, COM is proprietary. It is owned by a single vendor. While this might not be a problem per se, it does become a matter of concern when combined with Microsoft?s objective of Windows world-dominance. Specifically, COM’s viability outside of Microsoft’s operating systems remains an open ques-tion. Although Software AG has released COM for a number of non-Windows operating systems and Microsoft’s latest position on non-NT support for COM is that it will be supported "internally" as well, the viability and impact of this on the market is unclear. The strength of COM on Windows is the integration of COM with everything and anything that Microsoft does, as well as with third party's tools. This is not a strength on other operating systems. Microsoft's inter-operability strategy is limited to applications already deployed outside of Wintel platforms.

Although the client/server evolution clearly defined and delineated presentation, service, and business logic layers, the lack of server support tended to force business logic on the first tier. Further, systems that did exploit the second and particular-ly third tiers were noisy in terms of network traffic and did not perform well over low-speed, high-latency wide area networks.

Some early adopters of COM found that it did not scale. They also discovered that developing server-side application com-ponents required much more than just what COM alone could provide. Support for concurrency, database connection caching and pooling, efficient use of network connections, automatic object instance management, threading and data sharing policies, process isolation, and other facilities need to be in place before serious applications, supporting substantial numbers of concur-rent users, can be deployed in a real production environment. In addition, support for vital middleware features and services such as role-based and component level security, distributed naming, quality of service, load-balancing, failover, session concentration, synchronous and asynchronous transaction support, to name a few, need to be in place before COM can provide industrial--strength server support for scalable applications. This is where MTS and MSMQ come into the picture.

Now Does MTS Add Value to COM?

To put it simply, MTS is a development and runtime platform. Its general purpose is to provide a container for in-process COM server components, and to make it easier to write a scalable COM server that handles objects in a transaction environment.

MTS applications are implemented as in-process servers in COM-supported languages such is Visual Basic, Java, C++, etc. MTS acts as a container for these ActiveX components (see Fig. 2). The set of components that run in the same process is called a package. Each machine may have different packages running concurrently.

Packages provide an interesting level of control and diversity. The MTS administrator can change the property of a package to either "library" or "system." Components in a system package run together in the same MTS process (one per package). This process map, a DLL and its supported components into its address space along with the MTS Executive DLL, which pro-vides runtime services for components in the MTS process.

Library packages can be hosted in other processes. You could, for instance, configure MTS components to be loaded directly into Internet Information Server (IIS) processes. The most obvious reason for using library packages is the tight inte-gration between IIS and MTS, which allows each Active Server Page (ASP) to be run as a separate MTS-controlled process. This benefit is not without its price; library packages offer no component tracking, role checking, or process isolation.

Distributed transaction management within MTS applica-tions is the responsibility of the Distributed Transaction Coordinator (DTC) component of MTS. DTC was originally part of SQL Server 6.5, and provided two-phase commit (2PC). Now it is actually part of NT, not MTS. DTC coordi-nates transactions that span multiple resource managers by ensuring that a transaction's outcome is either commit or abort across all resource managers. DTC complies with OLE trans-action protocol, and with the Open Group's (formerly X/Open) XA-compliant resources such as DB2, Informix, Sybase, SQL Server, and others.

Microsoft provides administrative support with MTS Explorer, which will soon be a Microsoft Management Console (MMC) compliant snap-in GUI tool for administering MTS environments. MMC provides a console framework for management applications. One administrative issue right now is that while user accounts are integrated with NT security, and administrators can assign domain users and groups access to a package, component, and interface; the definition of roles, pack-ages, components, and interfaces cannot be controlled through the NT User Manager. Hopefully, this will be resolved with Active Directory Services in NT 5.0.

Components take on transaction support based on configuration values declared by the administrator, rather than by programmatic control. MTS provides four levels of transaction support value. A required component must execute within the context of a transaction. If the compo-nent's caller does not have a transaction context, NITS will I create one. A supported component can run transactional-ly or non-transactionally, depending on whether the caller has a transaction. Components that are not supported may not run in a transaction at all, for example, CreateInstance will fail. If a requires new transaction is called, MTS automatically creates a new transactions context object for the component, whether or not the caller already has a transaction context.

MTS provides other important Object Transaction Services, such as load balancing, Wolfpack support, thread pooling, con-nection caching, concurrency support, and Internet support. In short, MTS takes care of many intricacies which would other-wise be the responsibility of the COM component developer.

Is MTS Interoperable?

One problem for some enterprises is that many of these services and applications are not on Windows platforms now, but are on Unix and OS/390. Assuming that it is unrealistic to rewrite all of this for Windows in the near future, Microsoft?s goals must accommodate a middleware strategy, which can at least access these platforms.

Microsoft attacks this problem in two ways. One is through MTS's close integration with MSMQ, which, in turn, provides some support for and interoperability with other operating systems. Within MTS, however, Microsoft developed the COM Transaction integrator for CICS and IMS (COMTI, formerly known as Cedar) to wrap legacy trans-actions so that they appear to be typi-cal MTS components. COMTI (see Fig. 3) is shipped as part of Microsoft's SNA Server. It provides the capability to integrate COM-based applications with the legacy transaction processing systems that continue to run the majority of mission-critical applications in many corporations worldwide. COMTI provides not just the means of communication between the application servers running NT and the main-frame, but also development capabilities to "objectize" and to ?wrap" CICS and IMS transactions, so that they appear to be Automation objects to COM applications. Furthermore, COMTI provides automatic mapping of data types when the sending and receiving programs are written in different languages. For exam-ple, it will automatically map Visual Basic (or any other language that the COM component is written to) data types to Cobol data types.

How Well Does MTS Support Synchronous Transactions?

MTS allows developers to create COM components with sin-gle users in mind, and then to deploy them in multi-user envi-ronments. This is because MTS handles thread pools, concurrency, database connection caching, interoperability, and other services that are vital for a robust multi-user transactional sys-tem.

MTS provides many features and functions which indicate that Microsoft hopes to make COM a high roller at the enter-prise transactions table. Many have complained that DCOM needs adult supervision before it can provide enterprise-level support. Microsoft has provided an Object Transaction Manager (OTM) as a response.

The COM and MTS combination supports the synchro-nous programming model required by many applications. There is, however, another set of applications that include situations where information needs to be exchanged across application and sometimes even business domain boundaries, which must be supported by an asynchronous communication model. This is where MSMQ comes into play.

How Does MSMQ Support Asynchronous Transactions?

MSMQ (Microsoft Message Queue Server) is a Message-Oriented Middleware product that provides asynchronous messaging services. The disadvantage of synchronous commu-nication (MTS's strong point) is that the sender and die receiver are time dependent on each other, i.e., both need to be up and running at the same time. In addition, synchronous com-munication implies connection-ori-ented protocol where the sender and the receiver maintain a network connection. In asynchronous communication, the sender makes a request, but continues to execute while waiting for a response. Asynchronous messaging also implies that the receiver does not need to be up and running when the sender issues a request.

The most convenient way to handle asynchronous messages is with queues (see Fig. 4). Messages can be recovered if they are stored to disk. In addition, queue managers can sequence messages based on priority, time, or many other criteria. The receiver of the message call browse the queue, and select messages by criteria. The queue manager is also responsible for maximizing communications throughput between applications, and for performance.

Like any modern message queuing system, MSMQ has sup-port for handling undeliverable messages, as well as for recover-able and non-recoverable messages. Like MTS, MSMQ is administered through MSMQ Explorer, a snap-in which will soon be MMC-compliant, and is tightly integrated with Windows. MSMQ implements security based on public key technology and X.509, but does not support authentication or access control services. Like MTS, MSMQ uses Windows secu-rity, the Event Logger, and NT?s Distributed Transaction Coordinator (DTC), so that a transaction context can be shared across products.

MSMQ supports three types of configuration: dependent clients, independent clients, and full-blown MSMQ servers. Dependent clients rely on MSMQ servers to perform queuing operations, while independent clients can perform a number of queue operations autonomously. MSMQ servers allow access to enterprise and site information, and can route messages to target queues. MSMQ servers can also function as connector servers. It is this type of MSMQ server that allows MSMQ applications to communicate with IBM's MQ Series-based applications. This is an important feature, and it is strategic for those who need to interoperate with applications rising MQSeries.

MSMQ is a lightweight and fast messaging product, and the MSMQ API presents a non-object programming model. It moves messages, not objects. This is important for several rea-sons. Not all applications are object-oriented, particularly in legacy systems. MSMQ?s interoperability with IBM's MQSeries gives Microsoft the ability to reach out and communicate with any application running the MQSeries product, including thou-sands of CICS and IMS transactional systems that will most likely be responsible for running many mission-critical business applications for years to come.

Is This a Single-Vendor Best-of-Breed Solution?

On the distributed object side, COM provides an object model, while MTS manages and deals with many intricacies incurred by the core nature of the distributed model. MSMQ provides a number of Message -Oriented Middleware (MOM) services, including support for quality of service and support for the asynchronous, "fire and forget" type of processing. Microsoft?s aim is to bring it all together under one roof This is where COM+ comes in.

When it is released (Microsoft is planning to release COM+ together with NT 5.0), COM+ will provide new and enhanced support for marshaling, data-type handling, inter-face inheritance, the persistence model, and constructor sup-port. The primary purpose of COM+, however, is to provide a single object model across languages, enforced metadata, and, more importantly, common support for services provided by the operating system and by applications. These changes are related, since the solutions derive from a single architectural change.

The language problem with COM right now is for some languages, for example C++, developers must use COM-specific rules and API calls, such as CoCreateInstance and Release instead of the C++ new and delete. COM+ makes calls to the COM library easier by hiding the COM calls under C++ calls. For instance, programmers can use the C++ new to create an object - and the compiler generates CoCreateInstance. (Release is actually not needed under COM+ - it is handled internally.) The C++ compiler uses the COM+ library to resolve the C++ new.

In COM, objects may or may not provide metadata, depending on whether their interfaces are defined dynamically or statically. Objects are under no obligation to read the meta-data library to learn how to make calls. Under COM+ interfaces are defined with programming sequences (e.g., new instead of CoCreateInstance). The language compiler generates metadata, which is then stored in the object's binary file. Every COM+ object must have metadata.

This does not have much of an effect on Visual Basic (Microsoft already controls it) or Java (it already has standard interface constructs), but the effect on C++ will be interesting, since there is no standard way to specify interfaces. Of course, Microsoft will implement its own in Visual C++. But many tool vendors that support COM today and compete with Microsoft at the same time, will need to depend on the Microsoft-supplied COM+ library for compiling.

A more striking effect is that this new architectural wrinkle opens the door to a new type of client-object called an Interceptor. In COM, calls go directly from client to object. In COM+ they will pass through the COM+ library. Interceptor objects may be placed in the call path and will always be invoked automatically. Figure 5 shows two interceptors being used to provide load balancing and security. New versions of MTS will use interceptors heavily as a plug-and-play area.

What Is the Bottom Line?

Microsoft's Distributed Internet Application (DNA) architec-ture will be a service-rich framework for building applications based on COM+ and NT 5.0. Microsoft expects to spend over $85 million and employ more than 2,000 people to help ensure the success of independent developers in using this framework to develop functionally rich distributed objects.

Although it Is too early to say for sure, this cannot help but have a major impact on the overall middleware market. Corporations, after all, do not buy into infrastructure for the sake of having great technology. (What was wrong with OS/2 and Betamax? No applications.) The user community wants consistency, reliability, and a lot of application support. They also want service-rich middleware for a commodity price. Microsoft has made a major move in this direction by bundling COM, MSMQ, and MTS with the operating system, and by announcing COM+; and Microsoft seems poised to fulfill its major objective by giving the user community what it wants and what it needs. Of course, there are still a lot of challenges ahead for Microsoft-NT scalability and integration with legacy appli-cations, to name just a few, but it will be interesting to see how other middleware vendors, in particular CORBA vendors, will meet this new challenge from Microsoft.