Middleware

CORBA and Java: Marriage or just serious dating?

January 1999/By Max Dolgicer

Distributed applications can be developed using almost any product from the categories of middleware offered on today's market. These categories include database middleware that focuses mainly on accessing different data sources using ODBC and JDBC/vendor-supplied proprietary APIs, as well as message-oriented middleware (MOM), TP monitors, CORBA, MTS/COM and emerging Web-centric middleware.

While many IS executives are capable of figuring out when and when not to use database middleware, the situation is quite different on the rest of the middleware front. For many, choosing between typical procedural middleware technologies, such as MOM and distributed TP monitors, is a difficult one. Adding object middleware technology, such as CORBA and MTS/COM, to the mix only increases the difficulty in making a decision. Including Java -- a language and distributed platform that, in some people's estimation, replaces the need for middleware -- makes a decision close to impossible.

On the surface, CORBA and Java are two technologies that address many of the same enterprise network-based computing system design, development and deployment problems. Some people claim that these two technologies compete within the same market space. My opinion is that the two complement each other and will most likely live together in the enterprise, making use of each other's advantages while masking some disadvantages.

Several years ago, middleware products did not provide much more than a higher level abstraction of communication protocols. Using this approach, application developers could use a higher level API instead of low-level protocol interfaces, such as TCP/IP sockets or SNA LU6.2. Such an API masked the substantial differences between native APIs provided directly by communication protocols. Today, middleware's role has expanded from that of a communication vehicle to a comprehensive application development, deployment and management platform.

The middleware platform sits above such low-level plumbing as communication protocols and operating systems, and provides an extensive suite of services -- naming, security, load balancing and failover mechanisms, as well as reliable, flexible communication services. These platform services are an essential part of any serious effort to develop and deploy complex, mission-critical applications. CORBA and Java address the issue of standard interfaces to many middleware services without an allegiance to any specific operating system.

CORBA: The status report

CORBA is a specification produced by the Object Management Group (OMG), a Framingham, Mass.-based consortium comprising more than 800 companies. The OMG's goal is to specify an architecture that consists of an object request broker (ORB), common object services and vertical application domains. While the CORBA specification does not control how the technology is implemented, it does address the interoperation of implementations that are still maturing. Production systems, however, do exist. At the time this article was written, Version 2 of the CORBA specification was available. Major enhancements in Version 2 include: interoperability between products from different vendors; support for security and transaction services; language mappings for Ada, COBOL and Java; and a specification to bridge between Microsoft's COM and CORBA. Version 2 is still being enhanced, but the OMG is planning a commercial release of CORBA V3 in mid-1999.

CORBA V3 will address CORBA Messaging, as well as mechanisms and interfaces that allow an application to be assembled from parts -- similar to what the JavaBeans model and ActiveX hope to achieve. Messaging will be supported by a new scripting language and by the ability to pass objects by value. Some CORBA vendors, such as San Diego-based Expersoft Corp., already support the ability to pass objects by value even though it is not addressed in the CORBA V2 specifications.

Multiple interfaces per object will also be supported. This means that one object can provide multiple services to clients, or it can support several different versions of a particular service in parallel. It is interesting to note that multiple interfaces have been available with Microsoft's COM from the onset, whereas the OMG is just now catching up after several years of arguments about the merits of this concept.

Some additional enhancements have not been assigned to a specific category, such as a specification that will allow tools to be built for automatically generating CORBA IDL based on Java class definitions. Other specifications will define how Internet Interoperable ORB Protocol (IIOP) requests can be passed through a firewall, and how DCE-based legacy applications can be integrated with CORBA applications from both a client and server perspective.

Clearly, CORBA is more than just an ORB: It is a complete, distributed object platform. From the perspective of middleware service specifications, CORBA defines many of the services required to develop, deploy and manage distributed applications. These services include some of the "mortal" and must-have services, such as transactions, security, events, naming, persistence and life cycle, as well as some less-important services -- relationship, externalization and concurrency among others.

Many CORBA vendors are also working individually to add runtime and deployment services, as well as efficient policies for managing databases and TCP/IP connections, to their product lines. These services are the key to scalability, reliability and availability,

which is what middleware is all about. This makes some CORBA and CORBA-based products more than just a basic ORB that provides only object-to-object communication and support for location transparency. I would argue that implementing different middleware services is just as important as implementing different runtime and deployment services that are not associated with any standards organization. In fact, one of middleware's far-reaching goals is to
substantially reduce the amount of effort required to develop, deploy and execute applications in a multi-user environment.

However, some of these important features are still missing in most CORBA products, and either need to be developed in-house or purchased as a value-added solution. What is attractive about CORBA, is that it is a language-agnostic infrastructure -- the client and server parts of an application can be written in different programming languages. And thanks to the IIOP standard (CORBA 2), clients and servers can even be developed using different CORBA products.

CORBA interoperability

A client application residing on Vendor A's ORB can make a request upon a server object that executes under the control of Vendor B's ORB. Note that neither the client nor the server application is aware of the bridging technology connecting the two disparate ORBs.

There are essentially two types of request-level bridges: interface-specific and generic. Interface-specific bridges support predetermined Interface Definition Language (IDL) interfaces only, and are built with IDL-compiler generated stub and skeleton interfaces. Generic bridges can bridge requests to server objects of arbitrary IDL interfaces using the interface repository and other dynamic invocation support, such as Dynamic Invocation Interface (DII) and Dynamic Server Interface (DSI).

While interface-specific bridges may be more efficient, the requirement for prior compilation means that this approach offers less flexibility than the use of generic bridges.

ORBs can be connected to the IIOP protocol in two ways. First, they can use IIOP for all object communication -- connecting clients and servers within their own domain, as well as interacting with other ORBs. Second, an ORB can use another CORBA-compliant protocol or a proprietary protocol within its own domain to implement a so-called Half Bridge to interoperate through IIOP.

All vendors support interoperability with Microsoft's COM, a fact that cannot be overlooked when it comes to accessing CORBA objects from development and productivity tools. In addition, the forthcoming CORBA 3 specification includes support for important capabilities such as messaging, which had been the exclusive domain of MOM products. Support for additional services -- such as notification service, which was defined separately from CORBA 2 and CORBA 3 -- is further evidence that CORBA specifications and product implementations are maturing. I believe that the CORBA Messaging Service represents a special interest as it attempts to marry CORBA with MOM, which has proven to be a resilient technology.

CORBA Messaging Service

CORBA currently provides two modes of invocations: synchronous and deferred synchronous. With synchronous invocations, the client program is blocked until the results arrive. With deferred synchronous invocations, the client thread continues processing, subsequently polling to see if results are available. However, the latter requires the use of DII, which is somewhat cumbersome and still does not support true asynchronous mechanisms like callbacks. Because the word "messaging" is associated very closely with the words "asynchronous processing" and "decoupling," some might ask a legitimate question: "Why do we need CORBA Messaging Service if we have an Event Service defined?" The latter facilitates asynchronous processing, decouples clients from servers (actually, there are no clients or servers; instead, CORBA Events Service defines suppliers and consumers), and is now supported by all CORBA vendors.

The new Messaging Service addresses Asynchronous Method Invocations (AMI), which should affect only the client side. In most cases, applications are not required to change as from the server side. From a programmer’s point of view, all invocations can be treated identically, regardless of their synchronous characteristics.

AMI supports two models: callback and polling. In the callback model, the client passes a reference to a reply handler (a client-side CORBA object implementation that handles the reply for a client request), in addition to the normal parameters needed by the request. In the polling model, the client makes the request, passing in all the parameters needed for the invocation, and is returned as a Poller object that can then be queried at any time to obtain the results of the invocation.

By now users should come to an important realization: do not be fooled when the OMG uses the term "messaging." The CORBA Messaging Service does not provide users with a programming model like that supported by typical MOM products. You are still dealing with a client that invokes a method on a server object, which replies to this client. Compare this to the major MOM product, where one application writes a message to a queue and one or more decoupled applications retrieve the message from a queue. CORBA Messaging does provide some of the quality-of-service features typical in MOM products, including Delivery Quality, Queue Management and Message Priority Service (missing in CORBA Events Service), but does not address multicast for example.

Asynchronous messaging also introduces transactions to CORBA. Because client and server objects involved in a particular request may not be active at the same time, the concept of synchronous transactions breaks down. It has to be replaced with a set of transactions that each deal with the individual interactions between the client, the network, the server(s) and the actual work done by the server(s).

Routing is another new concept AMI introduces to CORBA. While this is transparent to application programmers, it is important to ORB vendors, who have to worry about the interoperability of their particular Messaging Service implementations. AMI allows for Time Independent Invocation (TII), which essentially provides support for: a) invocations on targets that are not currently active or activatable; and b) replies to clients that are not currently active or activatable. Routing supports TII through a store-and-forwarding mechanism, which implies the use of request routing agents.

Java: No introduction needed

Java has a lot of going for it and for good reason. As a language, Java has built-in threads and supports automatic garbage collection, a feature C++ developers value. In terms of support for object-oriented applications, Java supports not only classes but also the notion of interfaces, single-implementation inheritance and multiple-interface inheritance. Additionally, it provides support for complex and user-defined data types.

In terms of application development, Java facilitates superb productivity gains compared to other languages, such as C++. Putting aside the recent legal woes between Microsoft and Sun, it can be said that while there are some portability issues remaining on the client side -- especially in areas such as user interfaces -- they do not apply to the server side, where the famous slogan "write once, run everywhere" applies.

While major worries about Java performance remain, the Java Developer’s Kit (JDK) 1.2 incorporates some new features that should improve Java performance substantially. This does not mean that Java performance is currently inadequate, however. In fact, my experience is that Java performance is sufficient for many applications. Any conclusions about the applicability or non-applicability of Java are dependent on application requirements and service-level agreements. Nevertheless, help is on the way.

Java’s supported features go outside those of a traditional programming language, making it more than just another language. And Java as a distributed platform provides outstanding support for the development of Internet-based applications through the concept of applets.

Java applets vs. Java applications

The original intent of the Java applet technology is that applets would be small applications that ran within a Web browser. Its life cycle would begin when a user visited a Web page that contained an applet. After constructing the applet, a browser would then allocate a place for it within the screen’s viewing area.

Java applets leverage and take advantage of features that are built into Web browsers. This lets developers write applets, containing a rich amount of functionality, with a minimal amount of code. One of the major uses of applets is to display graphics and images. When incorporating GIF or JPEG files into a Java applet, it is not necessary to write any special decoding methods to interpret the image files. Instead, applets can use the browser's built-in decoder to display the images.

An applet has no life outside of an HTML page that is processed by a browser. The browser is the "container" of the applet, which depends on the browser to call its methods. There are many restrictions on the functionality of an applet; these are enforced by the browser and depend on the security options the user has selected.

There are four major types of restrictions imposed on applets:

- An applet has limited access to local and remote file systems.
- Applet classes loaded from the Internet are not allowed to execute any local commands.
- An applet cannot be a network socket server and may only open a socket connection to the server machine that served the applet's HTML page.
- Inter-applet communication is limited.

In contrast to Java applets, Java applications run without a browser. An application is invoked from the command line by passing the class name of the Java application to the Java runtime. Applications have complete access to the file system and network, and can also invoke native methods or code written in another language, such as C. This permits access to "legacy code from Java."

Above all things: Security

Most Java security revolves around the concept of a sandbox, which ensures that an untrusted application cannot gain access to system resources. To implement sandboxes, the Java platform relies on three major components: class loader, byte-code verifier and security manager. The class loader separates classes into namespaces and loads only the correct classes, while the byte-code verifier checks to make sure that classes are in the correct format, and can execute only within their namespace and memory boundaries. The security manager guards the execution of so-called "dangerous instructions" or code that tries to access protected system resources.

Java also has some inherent security features like type checking, which ensures that invalid type casting cannot occur. In addition, the Java Security API provides for digital signatures, message digests, key management and access control lists. With digital signatures, a supplier bundles Java code (and any related files) into a Java Archive (JAR). The supplier then "signs" the applet using a digital signature. The client can verify the authenticity of the supplier by verifying the signature. Message digests and keys can be used for encryption, while access control lists restrict access to services and resources.

Java RMI and more

The Java platform also supports features such as events, networking and database connectivity via JDBC. For example, publish/subscribe Java-based applications can be developed using events implemented in JDK 1.1. Java also has built-in serialization and reflection, which gives an application the ability to load Java classes dynamically and to look inside Java classes at runtime.

However, one of the Java platform's most important features is Remote Method Invocation (RMI). In fact, observers would argue that many questions related to the Java vs. CORBA debate have been triggered by the introduction of RMI.

RMI lets a Java program call a remote object once it obtains its reference. Conceptually this is very similar to the way CORBA supports distributed objects. But CORBA presumes a multilanguage environment, and must therefore have a language-neutral object model. In contrast, when RMI was introduced initially, it assumed the homogeneous environment of the Java Virtual Machine (JVM) and was concerned only with providing the foundation for Java-to-Java communications across virtual machines. Therefore, it could take advantage of the Java object model whenever possible.

As with CORBA, a Java RMI client invoking a method on a remote server object actually uses a stub or proxy for the remote object. This stub is an implementation of the interfaces of the remote object that forwards the request to the server object. The stubs are static in the sense that they have to be generated using a compiler before running the application.

The request is forwarded to the server-side skeleton, which is also static (compiler generated). The skeleton then makes an up call to the remote object, which carries out the actual method call. Java uses its built-in serialization mechanism to capture the value of parameters to the call
in order to send a copy to the server. It should be noted that this allows any Java type that can be serialized to be sent to the remote object, including user-defined objects. In other words, Java RMI supports passing objects by value, while CORBA can only pass objects by reference (this will change with CORBA V3). The RMI transport layer is currently implemented on top of TCP using Java sockets. While this is very similar to CORBA, there are two important differences:

- RMI does not support remote requests to be constructed dynamically -- i.e., without prior knowledge of the client -- but rather during its runtime based on discovery of the behavior of a particular remote object. This is implemented by the CORBA DII for clients, and by the DSI for servers.

- If a remote object is part of a replicated object, the client-side component in case of RMI can forward the invocation to each replica rather than to just a single remote object. CORBA does not provide this functionality.

For a client to invoke a method on a remote object, it must first obtain a reference to the object. While the CORBA Services specification includes a Name Service, the RMI system relies on the java.rmi.Naming class, which provides a simple way to obtain remote object references based on a Uniform Resource Locator (URL).

But how does a server know when an object can be removed from storage safely without causing a problem to clients that may still hold a reference to this object and might make another remote call on it? The answer is that RMI uses a reference-counting garbage collection algorithm similar to what Microsoft's COM provides. Interestingly enough, this flies in the face of the OMG and the anti-Microsoft alliance, which have been bashing COM's reference-counting mechanism for years.

A given server-based object may not be active when a request for it arrives. Therefore, both CORBA and RMI provide mechanisms for on-demand activation of objects. These are the Basic Object Adapter (BOA) and Activator, respectively. In CORBA, the BOA is responsible for passing requests to a server object through the skeleton (i.e., the server stub), the generation of object references, as well as activation and deactivation of servers. The Java Activator not only starts and initializes objects, it can also start entire JVMs if needed. Making a server object "activatable" requires some specific coding on behalf of the server application developer. In contrast to the client side, the server housekeeping code that must be included in the application is very different for CORBA vs. RMI-based servers. As a result, there is no way to develop server objects for RMI and later port them to CORBA or vice versa.

While the Java RMI provides the typical functionality of an ORB, it is not CORBA-compliant. It does, however, allow clients to make calls on remote objects using the same semantics as if these objects were local; has mechanisms to obtain a reference to a remote object; and activates objects that are not "alive" at the time they are requested.

Java: Stepping on CORBA's toes?

If you combine Java's RMI capabilities with the additional features and services discussed in this article you will know why some people now feel more lost in the world of middleware choices than ever before. Is it CORBA vs. Java, or CORBA and Java? Is Java middleware? The answers to these questions are not trivial. For IT executives, it is just another set of worries on the bumpy road to enterprise architecture.

Yet there is one piece of good news (although even this is subjective): In the forthcoming JDK 1.2 release, RMI will run on top of CORBA's IIOP. In addition, JDK 1.2 will include a pure Java ORB, and support a development environment that generates CORBA stubs and skeletons from Java IDL. Lastly, JDK 1.2 will be bundled with an all-Java version of CORBA Naming Service, thus providing at least an appearance that the war is over. However, it is premature to say whether these developments will bring peaceful coexistence, complete cooperation or something in between.

Until recently, OMG proponents had nothing to worry about. After all, the OMG defined and formalized interfaces to many vital middleware services that CORBA vendors have diligently started to deliver with their product releases. However, this might not be the case any longer.

Recently, the Java Division of Sun Microsystems Inc. released a set of specifications for many typical middleware services, such as naming, management, transaction and messaging. Java also has two component models, while CORBA currently has none. I will briefly explain some of these services; you can judge whether or not these services have any striking differences with the equivalent CORBA services. What is also similar is the approach that Sun's Java Division took. The focus in these services is on interface definitions and not specific implementation, thus allowing different vendors to supply different implementations of these services that all conform to the same interfaces. In fact, vendors can develop veneers that map Java Services APIs to existing implementations of these services, thus supporting the ability to call these services from Java applications and dramatically decreasing time-to-market.

Here is the brief description of some of these Java Services.

JAVA NAMING AND DIRECTORY INTERFACE (JNDI) -- By now everyone knows the importance of a Naming and Directory Service for distributed systems. The fact is that there are many interfaces and implementations of this service. JNDI is just another interface that provides naming and directory functionality to applications written in Java. JNDI provides methods for standard naming operations, such as associating attributes with objects and searching for objects using their attributes. What is important to understand is that JNDI is defined independent of any specific naming or directory service implementation. It therefore provides application portability using this kind of common API across different naming and directory services implementations.

JAVA MESSAGING SERVICE (JMS) -- The JMS specification provides developers with a standard Java API for enterprise MOM services, such as reliable queuing, publish and subscribe communication, and various aspects of push/pull technologies and quality of service. JMS supports the asynchronous exchange of data and events through a common API. While MOM technology is mature -- there are many applications in production and many vendors to choose from -- standards were never its forté. In fact, MOM vendors never pursued the issue of standards at the API or wire protocol level, compared to what the OMG did along the lines of CORBA IIOP. Assuming that vendors of MOM products adhere to the JMS specification, it would be possible to write portable, message-based Java applications at the API level at least.

JAVA TRANSACTION SERVICE (JTS) -- One of the major goals of JTS is to ensure that multivendor transaction processing solutions interoperate in a Java environment. The Java Transaction API specification consists of two parts: the first provides an API that Java client applications can use to define the demarcation of a logical unit of work or transaction that they want to control. The second part is important to those vendors that implement transaction manage- ment products (TP monitors) and resource managers (mostly DBMS and MOM products). It defines Java mapping to XA interfaces (The Open Group standard, which is based on the two-phase commit protocol).

Most JTS interfaces are Java mappings to the CORBA Object Transaction Service (OTS). However, JTS extends OTS in two ways: it incorporates the synchronization interface (also defined in the in-progress-draft of the upcoming OTS revision); and it specifies how a transaction context can be propagated through different communication protocols. In sum, JTS is nothing more than Java mapping of OTS interfaces -- good news for CORBA proponents.

JAVA MANAGEMENT API (JMAPI) SERVICE -- JMAPI is a set of Java classes and interfaces that build and deploy management applications. It can be used to build application and system management components
at various levels. JMAPI also includes extensions to the Abstract Window Toolkit (AWT), providing pre-built, high-level GUI components like meters and charts that can accelerate the development of a management console for distributed Java applications. The specification also addresses how to instrument individual Java server objects and applets, as well as Enterprise JavaBean (EJB) containers for monitoring and management. Finally, it deals with the protocol that is required to communicate management events within the system management environment, including integration capabilities for SNMP.

Java and the component world

So far we have looked at Java from the perspective of a programming language and a platform. We have also looked at various features, such as the ORB capabilities provided through RMI, and middleware services, such as JNDI, JMS, JTS and JMAPI. However, the Java specification goes much further: it includes a component model in the form of JavaBeans and Enterprise JavaBeans. The first is geared toward application development and is focused more on the client side, while the second model deals with the issues surrounding the deployment and management of server components.

What is a JavaBean? According to the official definition, "a JavaBean is a reusable software component that can be manipulated visually in a builder tool." In order for a piece of Java code to qualify as a Bean, it has to provide certain functionality in addition to its purpose as an implementation of, for example, some specific business function. This added functionality then lets Beans be assembled using graphical development tools. In particular, the Java-Beans spec addresses the following:

- Development tools can rely on the Bean to support the concept of introspection; for example, the tool can automatically analyze what functionality a Bean exposes.
- The Bean can be customized.
- Beans need to make invocations on other Beans. This is handled by a simple event model.
- JavaBeans, like any other object model, allow an object to maintain its state in some form of properties. The properties in the JavaBeans model are also used to hold the customization information.
- Finally, it should be possible to have persistent objects -- objects that can be saved to permanent storage and be brought to life again at a later point in time. The JavaBeans specification therefore defines a serialization mechanism.

In addition to this functionality that relates to the Beans themselves, the JavaBeans specification also defines how Beans can be packaged (by a vendor, one development team of an end-user company and so on) for delivery to developers who would use the Beans in their applications.

Enterprise JavaBeans

While JavaBeans are focused on the development and application assembly aspects of Beans, the Enterprise JavaBeans specification addresses the issues that revolve around the execution environment that hosts a Java application. There are many different execution environments available today, most notably TP monitors, DBMSs, Web servers and Web application servers, as well as CORBA-based systems. All of these try to provide scalability, load balancing, resource optimization, high availability and reliability, as well as support for a large number of concurrent clients. These issues distinguish the server side of a distributed application from the client side, and make the development of mission-critical enterprise servers a much more daunting task.

One of middleware’s major goals is to allow development of complex business applications with a single user in mind, while providing a services-rich, runtime environment that facilitates deployment of these applications in a multi-user environment. In an ideal environment, it is up to the middleware platform to take care of process and thread management, isolation and activation policies, concurrency, database connection pooling, lifecycle management and other complex issues. In short, middleware needs to provide an execution environment for enterprise applications.

Such environments exist today -- MTS/ COM and TP monitors such as Tuxedo, CICS and Encina are just a few examples. The drawback of these various execution environments is the fact that applications developed for any one of these environments are proprietary.

Furthermore, most of them are fairly complex, which means that users first have to develop a sophisticated server application. This also means that users will be stuck with the execution environment they have been coding to.

This is essentially what the Enterprise JavaBeans specification is trying to solve. Its goal is to provide an execution environment that gives users all of the required services needed for mission-critical enterprise server applications, without forcing them into a proprietary solution. Server applications that are developed based on Enterprise Java-Beans are supposed to be portable across any implementation of the specification.

Each server Bean lives in an Enterprise JavaBeans container. This container provides all of the Bean’s important execution services: state management, transactions, security and life-cycle management. The server Bean does not have to worry about such issues as how a client can locate it and when is the correct time to release memory.

The container automatically provides an EJB Finder that the client can use to locate the server Bean whose services are needed. If no instances of the server Bean exist, the EJB Factory creates one; finally, there is a representative (proxy) of the "real" server Bean that the client can invoke methods upon. This proxy Bean is like a shell around the real application Bean; for example, it allows the EJB server to introduce security checks and to deal with transaction context.

Conceptually, Sun’s Java Division has focused on defining APIs and left vendors to implement this interface. A couple of years ago, the OMG adopted the same theory. So, is Java middleware? And does it mean that CORBA is in trouble? The short answers are a qualified no to the first question, and no to the second question. So why is Java not a middleware platform, or at least not yet?

What is missing?

As outlined above, Java is evolving from a great programming language and distributed platform to a comprehensive middleware platform for distributed applications. Java’s initial focus on the client side based on small, downloadable applets is being expanded through Enterprise JavaBeans. This -- in combination with a number of middleware services like JNDI, JTS, JMS and so on -- is supposed to address the issues surrounding the development, deployment and management of enterprise-class applications.

However, all of these services are in their infancy. They need time to mature, and the problem is actually twofold: specifications for the different services are still evolving, and vendors must build "real" products that implement the specified services. The OMG has gained a lot of experience going down this path. The CORBA Core and CORBA Services specifications have matured over several iterations, and there are a variety of vendor products users can choose from.

The vendors who have produced (more or less) CORBA-compliant products have also gone through a learning curve. Today, most of their products are starting to deliver different middleware services as extensions to their basic products. Furthermore, some products are starting to address the vital services that have nothing to do with any standards -- load balancing, smart activation and resource management to name a few. These services have nothing to do with any language or standards.

Another crucial shortcoming of Java as a middleware platform is its singular language support. A homogeneous environment only exists in information technology utopia. A quick reality check calls for interoperability between distributed components written in Java, C, C++ and mainframe COBOL at a minimum. This is where Java, and in particular RMI, falls short and CORBA proves superior.

An important side effect of support for multiple programming languages becomes painfully obvious when legacy systems need to be integrated with new, Web-based applications. Legacy applications are not written in Java; for the most part they have been developed using COBOL and C. Theoretically, these applications might be encapsulated with Java code, but the multilanguage support in CORBA offers a more pragmatic and less-restrictive approach to wrapping legacy systems. While Java needs a comprehensive distributed object infrastructure, CORBA is capable of providing at least some of the infrastructure. This is exactly what CORBA and other middleware products add to Java.

So why is CORBA not in trouble? The short answer is Java. Not many observers will argue that the emergence of Java, as well as Web- and Java-based component specifications, supplied CORBA with the boost it needed.

One of the advantages that Microsoft's COM has over CORBA is the fact that Microsoft started with a desktop focus that made COM available basically everywhere. In contrast, CORBA's focus was more on the network and server side. This alone prevented CORBA from becoming middleware that could be used everywhere, at least initially. Add to this the lack of rapid application development (RAD) tools that could ease the pain of dealing with the intricacies of different aspects of CORBA-based development and you can see why CORBA products are still not in the hands of the masses.

Microsoft also has a well-defined component model with support for binary standards (COM), and a runtime environment for component execution (MTS). In contrast, CORBA still does not have a component model for either the client or the server side.

Java's arrival has changed the technology landscape substantially. First, CORBA will adopt the Java component model. As of now, forthcoming CORBA component model specifications will most likely be focused on client components rather than server components, but this will most likely change. Second, it is fair to say that CORBA's popularity received a substantial boost from Java's close association with Web-enabled applications. Java applets, Java applications and the positioning of IIOP as the lingua franca protocol of the Internet repositioned CORBA as a serious middleware contender for Web-based applications. Suddenly, CORBA products that support Java bindings found themselves as alternatives for the development of serious, Web-based applications.

The inclusion of Java ORBs in products such as Netscape's Communicator, Web Enterprise Server and, most recently, the Netscape Application Server and NetDynamics Application Server, have boosted CORBA's pervasiveness. Finally, as opposed to CORBA-based C++ applications, we can now write more portable server applications and move them from one system to another.

CORBA and Java add a lot of value to each other. But while the combination of CORBA and Java is such an attractive proposition, it is not love at first sight. Will there be a marriage between the two technologies? Let's just say that for now CORBA and Java are seriously dating. Whether they will bow to the inevitable, and whether this marriage will be a happy one, only time will tell.