WHAT IS MIDDLEWARE?

The Internet is the ultimate distributed system. Internet developers cannot be sure what types of hardware and software their customers will be using when they seek access to company servers. Internet applications designed to let one company interact with other companies can be complex because of the variety of hardware and software with which they must be able to work. Furthermore, building a complete, end-to-end integrated system is a complex task, regardless of the tools available. Most applications do not support integration sufficiently and require workarounds. More sophisticated applications need middleware to link application modules developed in different computer languages and running on heterogeneous platforms. To accomplish this linkage, companies must use distributed middleware systems.

Components are modules of code that can serve as the building blocks for even larger programs. Distributed components offer two advantages over traditional one- or two-tier programs in helping to integrate disparate systems. First, they provide common interfaces for the applications that need to be linked to build Web applications. Second, because it shifts the labor from written code to assembling prebuilt modules, the use of components can significantly reduce the cost and time involved in developing applications. Many companies already rely on the use of small components for the rapid development of user interface screens and for tasks such as linking applications to databases. Emphasis, however, remains on larger components and portfolio assembly. Larger components can offer functionality similar to that of off-the-shelf software, and portfolio assembly involves integrating multiple applications from different vendors into a suite of the best ones.

Scaling up from component-based software, middleware integrates code written for heterogeneous systems. The most popular of the standards for distributed component middleware are Microsoft’s Distributed Component Object Model, Sun’s Enterprise JavaBeans, and the Object Management Group’s Common Object Request Broker Architecture. An increasingly popular middleware option is the Extensible Markup Language (XML) coupled with a transport protocol like Simple Object Access Protocol (SOAP).

Development can be simplified if a company chooses to use a distributed component system with a component-based application server. Component Object Model or Enterprise JavaBeans application servers provide both basic support for a distributed component model and a wide range of integrated servers and utilities that the developers otherwise would have to create. For similar reasons, vendors are introducing Extensible Markup language servers.

Middleware comprises two general subsets: the application server or component environment, and messaging. Systems that maintain links between client/server systems or between applications in conventional distributed systems are examples of the former. The latter applies to all types of messaging software, including XML. Middleware is any system or collection of utilities that links different elements together, whether on a single machine or over a network.

When applications were maintained on mainframes, the operating system and its utilities tracked modules and passed messages with ease. Managing the communications between the computers and the database server was also relatively straightforward in later client/server systems that simply linked computers to a single database running on a database server.

Specific interfaces between the client and server for each application, which maintained the software modules that the servers needed to reach, made sense for mainframes and early client/server systems. Today, application developers rely on middleware systems that are independent of specific applications or components that might want to talk to one another. This independence is necessary because the developer can no longer anticipate the location, or platform used at each of the nodes in the distributed system.
With the rise of the Internet, dealing with the problem of disparate systems has become more urgent. Companies are building applications to be linked by the Internet to clients and to other servers throughout the world. The company building the application cannot know which kind of platform the client is running or where it is located. This problem will only become more complex as a variety of mobile wireless devices begin to access company sites via the Internet.

In the world of networked computing, middleware must link a variety of platforms from an unknown number of locations. The middleware that keeps track of the locations of the software modules that need to link to each other and that manages the actual exchange of information is rapidly increasing in sophistication. In short, middleware is being used to solve the two distinct problems of heterogeneity and resource discovery.

**TYPES OF MIDDLEWARE**

A major challenge in building applications that can be partitioned over distributed systems is the creation of a uniform scheme for passing information among program modules and for accessing data from multiple sources when programs and data may reside on different platforms. Middleware provides developers with a uniform interface through which their programs can access other applications. Each application interacts only with the middleware, which performs the necessary translations to communicate with the appropriate databases, operating systems, and applications. The middleware thus presents the application developer with a single, consistent interface (the illusion of a single underlying server) that masks the complexity of the actual computing infrastructure.

Several types of middleware are often used together, because no single middleware approach is general enough for the full range of applications. Therefore, systems are not uniform and single-server, but are collections of subsystems that use a variety of middleware to interoperate. This diversity removes the pressure for an entire enterprise to use a single solution and encourages business data standards for integrating middleware systems. In other cases, heterogeneity is unavoidable, so bridging integration solutions are necessary.

In addition to providing connectivity, middleware often provides special services for an application when it is desirable to isolate those services rather than build them directly into the application. Isolation is beneficial if a service is used by multiple applications (saving money by providing a service all applications can share) or if the software to implement those services is purchased from a middleware vendor. Some of these services function independently, and their association with true middleware is merely historical. (Some common examples are Common Object Request Broker Architecture’s collections, properties, query, relationship, and time services. Although all these are application independent, none require intimate integration with the middleware infrastructure.) Other services, however, are intertwined with the middleware’s connectivity services (such as security, high availability, and load balancing).

Of greatest interest to developers of distributed systems is middleware for remote procedure calls, messaging, transaction processing and object and component management, as well as the dominant types of distributed middleware, such as enterprise application integration (EAI), distributed components, and XML.

**Remote Procedure Call (RPC) Middleware**

The RPC middleware model is based on a synchronous approach to communication. When a procedure has initiated a call to another procedure, it waits for the response. If a call from procedure A to procedure B initiates subsequent calls from procedure B to procedure C, etc., each procedure involved in the sequence waits as the call is transmitted out and back. (In **synchronous** communications, data are transmitted at regular intervals and in one direction. In **asynchronous** communications, data can be transmitted at any time and in any direction.)

The primary drawback to RPC middleware is that its programming model is at a low level of abstraction. Consequently, applications using RPCs are tedious to write. RPCs allow point-to-point communication between machines, but they do little to mask the
details of implementation from developers. Each RPC in an exchange, as well as control
flow embedding the RPCs, must be explicitly coded, which slows development and re-
sults in code that is more difficult to change than it would have been if components and
objects had been used. Another disadvantage of this type of middleware is that it works
for applications that use few RPCs, but in high-volume systems or in cases where the
function call will take a long time to complete, an asynchronous approach works better.

**Message-Oriented (MOM) Middleware**

MOM is a popular asynchronous model for exchanging messages between applica-
tions or modules. A MOM environment incorporates a queue. The initiating module
sends a message to the queue, then goes on with other processing. The queue tries to
send the message to the target module, continuing to do so until the target module is
available and accepts the message. Once the target module has created a response,
that response is also sent to the queue, which then forwards it to the originator, again
continuing to forward the message until it is accepted.

Queues are especially effective if the originator or the target modules receive an
overwhelming number of messages. In such cases, the queue mechanism holds the
messages and forwards them only when the target module can process them. In that
way, MOM prevents the source from blocking and prevents the target from being
overwhelmed. MOM also can add at least a second target process to read from the
same queue, increasing the rate at which the messages are processed.

A common approach is to use commercial MOM software, which often provides
functionality beyond simple message passing. For example, most MOM systems incor-
porate some way of keeping track of where modules are so that the target can be
more or less transparent to the sender. Similarly, if a MOM system is to pass messages
between modules written in different languages, it requires a neutral Interface De-
scription Language (IDL) mechanism to provide a common language for the two
modules. MOM systems have been developed to handle asynchronous calls between
procedures, between components, or between modules or applications.

**Transaction Processing Middleware**

In database systems, a *transaction* represents a group of commands that must succeed or
fail as a single unit. An example is an operation that transfers funds from one bank ac-
count to another. Both halves of the transaction must occur before the transaction can
succeed. If funds are debited from one account, they must be credited to the other ac-
count. If one of these operations fails, all parts of the transaction must be restored to their
original states. That is, a transaction processing system keeps a copy of the original state of
the elements in the transaction and releases that copy only when all changes necessary to
the transaction have been accomplished. Transaction processing systems are essential to
financial systems and to any other applications that require transactional integrity.

Transaction processing monitors, also called transaction managers, are based on
mainframe processing. Examples include the Customer Information Control System
(CICS) and Information Management System, which have matured to provide excel-
lent performance and reliability.

Modern distributed computing applications frequently use middleware to access
relational database servers using Structured Query Language (SQL) (discussed in
Technology Guide 3). Ideally, business software applications would be written as a se-
ries of transactions. All participating application components would adhere to the
transaction-management paradigm. In reality, there are two basic approaches to
transaction management:

1. **Without a separate transaction-processing monitor.** This approach uses the trans-
action management capabilities of the relational database management system’s
server product.

2. **With a separate transaction-processing monitor.** This approach is required when
several heterogeneous databases are being updated in a single transaction. It also
can improve performance in high-volume transactional applications.
The three best-known component models are Microsoft’s Component Object Model (COM), Sun’s Enterprise JavaBeans (EJB) model, and the Object Management Group’s (OMG’s) CORBA Component Model (CCM). The component level comprises interfaces and an environment with services that keep track of components and pass messages between them. Each of these three component models (CORBA, COM, and JavaBeans) has been extended by the creation of a new, more advanced component model (the new CCM, COM+, and EJB, respectively). One of the advantages of component software is that its development is independent of how the component is written. The only attributes of a component that a developer needs to know are its interfaces.

**OMG’s CORBA**

The CORBA system is transparent to the developer in that he or she does not need to know where the target object is located. It can reside on the same machine or on another platform with the same or different operating systems. To use CORBA, developers wrap components written in a specific object-oriented language by creating IDL interfaces. The code that implements a CORBA environment can be object-oriented code, but it also can be written in a non-object-oriented language like C. At the CORBA or component level, one does not care what is inside the components. Thus, CORBA can also be used to wrap COBOL modules and integrate them.

**Microsoft’s COM**

Microsoft’s COM is another approach to components that defines an environment that allows a developer to create interfaces for code modules, then allows any code modules to send messages and obtain results from other COM modules. COM modules are not objects. They lack some of the characteristics that have traditionally been used to define objects, specifically inheritance. COM modules are components. They have interfaces defined in Microsoft IDL and an environment, which is built into the various Windows operating systems, that passes messages between COM components.

To pass messages to COM components on networked Windows platforms, a Windows developer uses DCOM, which is an RPC mechanism that moves messages between distributed COM components. COM components can be developed in non-object-oriented languages like C. Like CORBA, the key is that at the component level, the system is simply concerned with interfaces and passing messages and does not concern itself with what kind of code is inside the COM components.

**Sun’s Enterprise JavaBeans (EJB) Model**

This component system was developed by Sun to provide its Java object-oriented language with modules that would be easy to reuse. Sun defined a specific type of Java object that it referred to as a JavaBean. JavaBeans have interfaces written in Java. Sun also specified utilities (other objects) that would manage message passing between JavaBeans. Java’s Remote Method Invocation (RMI) enables a Java program running on one computer to access the objects and methods of another Java program running on a different computer (i.e., it is an RPC-like mechanism for accessing JavaBeans on distributed platforms).

**Enterprise Application Integration (EAI) Middleware**

EAI-packaged middleware is designed to allow developers to choose different levels of integration (e.g., data-level, application interface-level, or business process-level) between existing and new systems. As a result, many older systems that still perform valuable tasks are able to continue. Indeed, often the return on an EAI investment is measured by how little of the installed base needs to be changed to leverage a new system.
Another driver behind EAI deployment involves sharing information between disparate applications that never were designed to work together. Business process automation is the automation of tasks that an organization already might be doing manually by integrating two or more applications. For example, a business can print a sales report from an ERP system to cross-reference customer information for credit checks in another system. EAI would transform this task from a manual process to an automated one.

The single-application vendor solution promises the highest degree of integration among applications. However, it is unlikely that this approach can satisfy requirements for complete integration across a portfolio of applications because, according to various analysts, packaged applications address only about one-third of an enterprise’s requirements. The remaining two-thirds are satisfied by custom applications. So, once integration spreads beyond the scope of a single application suite, planners, designers, and architects must rely on EAI middleware. EAI middleware addresses the uppermost three layers of services seen next.

**DISTRIBUTED COMPONENT MIDDLEWARE**

Although components are often used as a necessary part of middleware, they are not required. Middleware systems have been built in a variety of ways, using different technologies. As the complexity and heterogeneity of the typical enterprise’s portfolio of applications increase, however, distributed component middleware systems may play a role in reducing that complexity.

A potential approach could be to convert all of a system’s modules into independent components with standard interfaces. Then, instead of wiring the entire system together, a middleware system could be used to pass messages between the components. Because the components would be uncoupled, they could be added or eliminated without changing the entire system. However, the standard interfaces are not currently provided by most widely used component models. Most commercial middleware packages use the MOM model, and each package has a proprietary method for connecting an application to the middleware. Open standards, such as the Java Cryptography Architecture, are just beginning to come into use, and no major application vendors are supporting them. Current middleware connectors are makeshift, customized components that use the application’s native API to get the interface to work. Some vendors are starting to standardize the connections between applications and middleware, as Siebel is doing with its XML-based EAI bridge.

Developers can reduce the complexity of developing distributed component systems in other ways. Many middleware systems implement protocols, such as APIs, and supply a few utilities, which are usually components themselves. If the information passing between two components running on different platforms is uncomplicated, then a minimal middleware package can be used.

However, many distributed applications and almost all e-commerce applications require more than simply passing data between two modules. For example, a distributed application may need a transaction-processing model that ensures that updates to data are not finally committed until all aspects of the transaction are ready to be applied. Most Internet systems need the capability to scale from one to multiple servers. This task requires an application designed in such a way that multiple instances of it can run without producing invalid results. It also involves load balancing to ensure that the server is not overloaded during peak periods of activity.

**Distributed Component Architectures**

Software architectures describe the basic elements and relationships used in software development. Companies that embrace distributed component middleware are adopting a component architecture. In such cases, the key elements in the business environment are described in terms of components, and a middleware infrastructure is included to facilitate communication between the components. The three most popular component architectures are OMG’s CORBA architecture, Microsoft’s Distributed Internet Architecture (DNA), and Sun’s J2EE architecture.