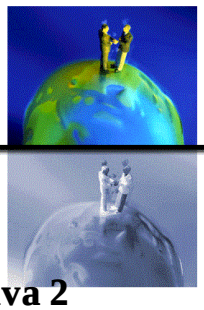

Professional training
and consulting in Enterprise JavaBeans (EJB) and Java 2 Platform, Enterprise Edition (J2EE) Technologies

The Technical Benefits
of EJB and J2EE Technologies over COM+ and Windows DNA

By Ed Roman and Rickard Öberg
December 1999

Prepared for Sun Microsystems, Inc.

http://www.middleware-company.com

Table of Contents

About This Whitepaper

This whitepaper explains the technical benefits of J2EE and EJB over Windows DNA and COM+. For a business analysis of these two architectures, please see the accompanying whitepaper, *The Business Benefits of EJB and J2EE Technologies over COM+ and Windows DNA.*

This whitepaper was primarily written by two authors, however we would like to thank our independent review panel that has given us feedback, contributions, and technical suggestions. This includes Ian McCallion, Robert Orfali, Gopalan Suresh Raj, Anne Thomas, Karl Avedal, Jonas Wallenius, Doug Hibberd, Scott Brittain, Chip Wilson, Adam Berman, and Floyd Marinescu.

Executive Summary

In today's world of complex enterprise information systems, corporations are realizing faster time-to-market and lower development costs by purchasing their middleware from a server-side platform vendor. Such a vendor provides a robust underlying commerce platform that enables developers to craft high-performing, scalable, maintainable, and multi-user secure commerce systems.

Today, there are two prominent choices for a server-side platform. The first choice is Sun Microsystems' Java 2 Platform, Enterprise Edition (J2EE), which contains the Enterprise JavaBeans (EJB) server-side component architecture. The second choice is Microsoft's Windows Distributed interNet Applications Architecture (Windows DNA), which contains the COM+ server-side component architecture.

J2EE platform-based products offer the following technical benefits over Windows DNA:

Component types supported. Support for both business process components as well as data components is critical if one is to build a complex and re-usable domain object model. Windows DNA has no support for data components, whereas J2EE platform-based products support both types of components today.

State management. Underlying platform support for stateful business processes can simplify code and actually enhance a deployment's scalability. Windows DNA has very poor support for a stateful business process component, and does not provide state management services. J2EE platform-based products support stateful business processes today.

Database caching. A robust data caching system enhances the scalability of a commerce system significantly. Microsoft's strategy of caching database data in the middle tier was formerly part of the In-Memory Database (IMDB), which has been permanently removed from Windows 2000 due to functional limitations. This leaves Microsoft without a compelling data caching story. By way of comparison, several J2EE platform-based products support middle tier caching in a robust manner today.

Declarative (automated) persistence. A declarative persistence model enables developers to persist enterprise data without writing code, such as Java Database Connectivity

(JDBC) code. This reduces coding time significantly, and allows one to author database-independent code. Windows DNA also does not support declarative persistence. By way of comparison, several vendors implementing the J2EE platform support complex automated persistence today.

Scalability through load-balancing. For a scalable multi-tier deployment, a corporation must be able to add middle tier machines as necessary to handle high-volume transactional load. This requires logic to automatically load-balance across middle tier components. Windows DNA currently does not load-balance client requests across middle tier components. Several J2EE platform-based products do so today.

J2EE Overview

The Java 2 Platform, Enterprise Edition (J2EE) was designed to simplify complex problems with the development, deployment and management of multi-tier enterprise solutions. J2EE is an open industry standard, and is the result of a large industry initiative led by Sun Microsystems. Examples of vendors that are part of the J2EE collaboration include IBM Corporation, BEA Systems, and Oracle Corporation. These vendors are part of a larger group of over twenty-five (25) vendors that are implementing J2EE platform-based products, all to the same industry standard.

Because J2EE is an open standard, J2EE promotes choice of vendor products and tools, and promotes best-of-breed products through competition in the middleware and tool space. The cornerstone of J2EE is Enterprise JavaBeans (EJB), a standard for building server-side components in the Java programming language. J2EE also supports cross-platform development, such as development on Sun's Solaris Operating Environment, Linux, IBM OS/390, or Microsoft Windows. This is possible because J2EE rests on the platform-independent Java virtual machine.

The origins of J2EE trace back to April 12, 1997, when Sun Microsystems announced an initiative to build the Java Platform for the Enterprise (JPE). The JPE consisted of a suite of standard Java extensions, known as the Enterprise Java APIs. Examples of the Enterprise Java APIs include Enterprise JavaBeans (EJB), Java Servlets, and the Java Naming and Directory Interface (JNDI) Technologies.

The goal of the JPE was for any middleware vendor to implement a standardized execution environment for distributed enterprise applications, either on top of their existing, proven middleware solutions or as part of new emerging products. An application developer writing to the Enterprise Java APIs would gain a platform-neutral and vendor-neutral interface to server-side programming.

The JPE services have been extremely successful, and currently there are over twenty-five (25) products that implement the EJB specification alone. However, Sun Microsystems still had several challenges to overcome. The foremost problem with the JPE was there was no way to test whether a server-side platform did indeed comply with the JPE specifications. Another problem was that each Enterprise Java API evolved separately; they were not unified with locked-down version numbers. This forced developers to manage the integration between the different API releases. Finally, there was no default reference implementation that developers could use to build their application code.

Sun Microsystems has answered these challenges by releasing the Java 2 Platform, Enterprise Edition (J2EE), which is today's evolution of the JPE. J2EE consists of the following deliverables:

J2EE Platform Specification, a definition of the underlying platform that a middleware vendor must implement to have a J2EE platform-based product. The J2EE Platform Specification synchronizes the releases of the various Enterprise Java APIs, yielding a single unified platform for server-side development. It also provides tighter integration between the various Enterprise Java APIs. Finally, it provides for interoperability with existing systems, such as CORBA-based systems.

J2EE Reference Implementation, a complete implementation of the J2EE technology specifications. The J2EE Reference Implementation demonstrates the viability of the J2EE platform, and can be used to develop portable J2EE applications.

J2EE Compatibility Test Suite, a suite of tests that a product must successfully pass to receive the J2EE certification logo. The test suite is the key to J2EE application portability. It ensures development of applications that are portable across a variety of J2EE platform-based products. It also assures developers that any J2EE platform-based product will support the full Enterprise Java API suite.

J2EE Blueprints, the programming model for J2EE applications.

J2EE Object Model

Figure 1 depicts a typical three-tier J2EE deployment. Note that a complex system spanning corporate boundaries may have more than three physical tiers, and a simple system may have fewer tiers.

The following is a break-down of each tier. The presentation layer contains components dealing with user interfaces and user interaction, such as rich graphical user interface code or flavors of HTML code. The business layer contains components that work together to solve business problems, such as logic to bill credit cards or process orders. The data tier contains any databases, existing enterprise information systems, legacy systems (such as a CICS system running on MVS), or an enterprise resource planning (ERP) system such as SAP or PeopleSoft.

In J2EE, the presentation tier can include CORBA clients (clients written in languages other than the Java language), Java applets, Java applications, Java servlets, JavaServer Pages (JSP), and static Web pages. CORBA clients use the CORBA Naming Service (COSNaming) to locate middle tier components, and use CORBA/IIOP to invoke methods on those components. Java clients use the Java Naming and Directory Interface (JNDI) to locate middle tier components, and RMI-IIOP to invoke methods on those components. Although it is not shown in Figure 1 (for clarity), messages can also be sent asynchronously using the Java Message Service (JMS).

The J2EE business tier contains business and data logic. The Enterprise JavaBeans (EJB) architecture is the server-side component model for encapsulating such logic. EJB components currently include session beans (business process components) and entity beans (data components). When a client invokes a method on a component, the J2EE platform-based product intercepts the call, and delegates it to the component. At this point of interception, the J2EE platform-based product can perform a variety of middleware tasks, such as transactions, state management, security, or persistence. To integrate with non-Java code, one can either wrap that code in an EJB component that delegates to it via the Java Native Interface facility or wrap that code in a CORBA shell, and delegate to it via Java Interface Definition Language (Java IDL).

To integrate with databases, Java Database Connectivity (JDBC) can be used, or SQL/J can be used to interlace Java code with SQL. Integration with existing systems is performed via proprietary means, and will be standardized in the future via J2EE connectors, which are planned to become part of a future J2EE platform release.

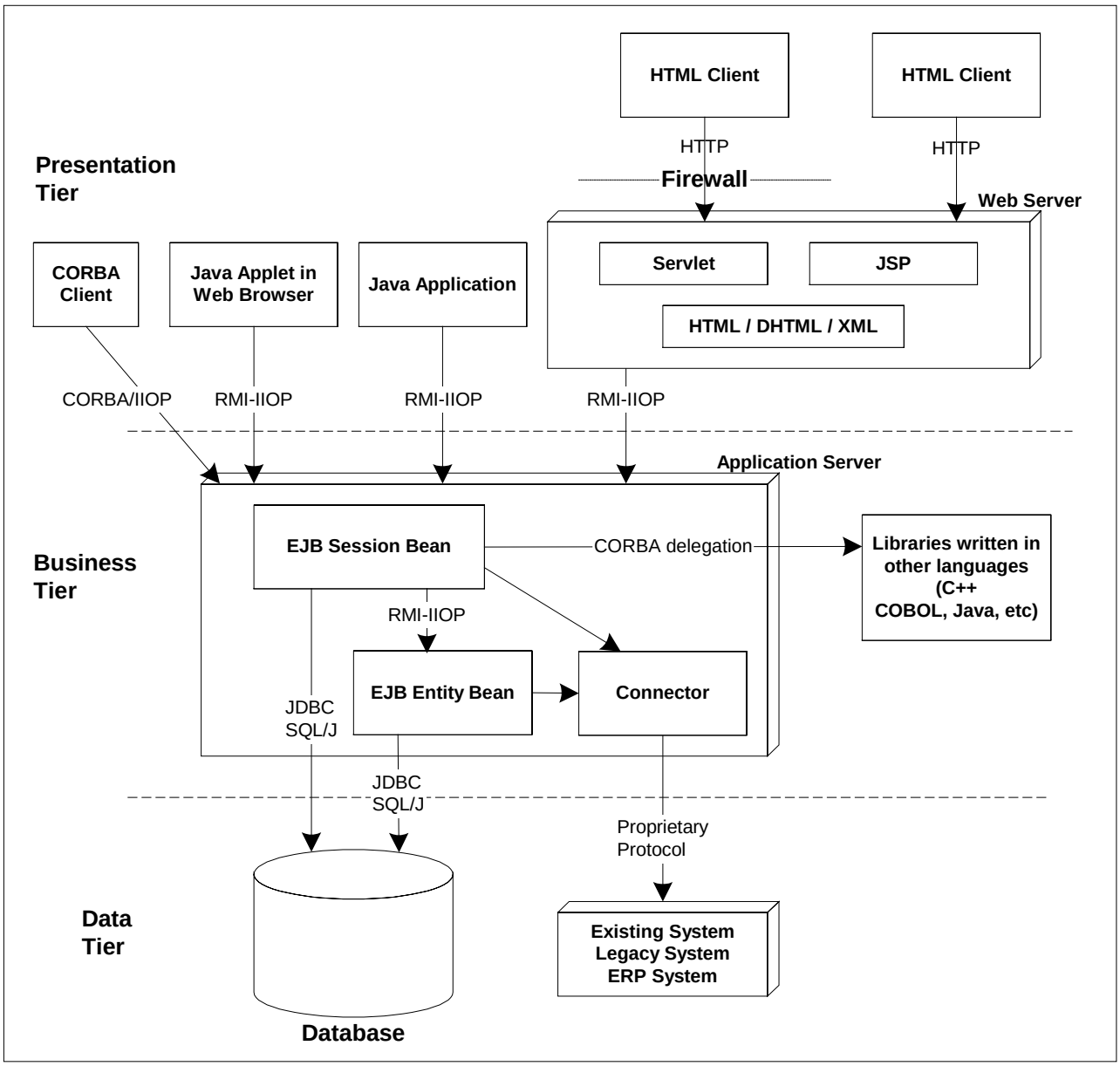

Figure 1: The J2EE Object Model

Table 1 is a summary of the J2EE Enterprise API specifications.

Enterprise JavaBeans (EJB)	Architecture for building re-usable server-side components. Defines the rules governing EJB containers and the components that reside within them.
Java Database Connectivity (JDBC)	Java technology-based relational database interface, allowing access to any relational resource that has a corresponding driver, such as Oracle, Informix, and SQL Server.

Table 1: The J2EE API suite (*continues*)

Java Naming and Directory Interface (JNDI)	Used to locate resources over the network, such as EJB components, database drivers, and security credentials.
Java Remote Method Invocation over the Internet Inter-ORB Protocol (RMI-IIOP)	The RMI-IIOP interfaces enable method invocations across Java virtual machines. If an IIOP protocol is used, J2EE can integrate with non-Java code written to the CORBA standard, such as C++ or COBOL code.
Java Message Service (JMS) *(JMS support is optional in the first J2EE platform release)*	Enables asynchronous communications, including point-to-point and publish/subscribe messaging.
Java Interface Definition Language (Java IDL)	Java technology-based CORBA ORB implementing a subset of the CORBA specification suite.
Connectors *(slated for a future J2EE release)*	Provides access to existing enterprise information systems, such as CICS, Tuxedo, SAP R/3, and Peoplesoft.
JavaServer Pages (JSP)	Technology that allows Web pages to be dynamically generated. Enables Web designers without serious programming knowledge to leverage middleware.
Java Servlets	Similar to JSP technology, servlets are request/response oriented components that are typically deployed in a Web server. They require Java knowledge, and are typically used to manage session state and dynamically generate HTML.
Java Transaction API (JTA)	Used to demarcate transactional boundaries programmatically.
eXtensible Markup Language (XML)	The Java XML API provides an interface to an XML parser and a set of commonly used methods for manipulating XML. Used to describe EJB components, and as a file format for JSP scripts.
JavaMail	Used to send electronic mail, such as an order confirmation.
JavaBeans Activation Framework (JAF)	Enables the automatic activation of the appropriate classes needed to manipulate various types of media which might be included in a mail sent via JavaMail.
Java 2 Platform, Standard Edition (J2SE)	J2SE (formerly known as Java 2) is the underlying Java platform that rests beneath J2EE. J2SE includes core services such as the *java.lang* and *java.util* libraries.
Load-balancing, data caching, transparent failover, and other enterprise services	Services such as these do not require a specification, as a middleware vendor typically provides these qualities of service transparently, without affecting application code. J2EE was explicitly designed to allow vendors to differentiate themselves in these areas by offering varying qualities of service, benefiting the customer with best of breed solutions.

Table 1: The J2EE API suite (*continued*)

Windows DNA Overview

Like J2EE, Microsoft's Windows DNA was designed to simplify complex problems with the development, deployment, and management of multi-tier enterprise solutions. The most significant difference between Windows DNA and J2EE is that Windows DNA is a proprietary product supported by a single vendor, whereas J2EE is an open industry standard supported by a variety of middleware vendors, each of which are providing implementations for the standard. The cornerstone of Windows DNA is COM+, a language-independent technology used to build re-usable components.

Windows DNA has evolved from the middleware services provided in Windows NT. These include clustering services, Web component services, and development and management tools. COM+ has evolved from several Microsoft products: the Component Object Model (COM), Distributed COM (DCOM), Microsoft Transaction Server (MTS), as well as parts of Microsoft Message Queue (MSMQ). Today, COM+ includes several middleware services beneath its hood, including transaction management, resource management, and security management.

Windows DNA applications realistically must rest on the Windows 2000 operating system. There are several Windows 2000 packages:

Windows 2000 Professional targets desktop and laptop users.

Windows 2000 Server targets file servers, print servers, Web servers, and entry-level application servers. Windows 2000 Server adds the core Windows DNA services.

Windows 2000 Advanced Server targets larger Web and application servers. Advanced Server adds additional logic to spread traffic across multiple Web servers.

Windows 2000 DataCenter Server targets large Web servers, application servers, and data servers. DataCenter Server adds enhanced failover support.

Each of the above Windows 2000 packages builds on the previous package, and adds support for more processors. Microsoft may release future versions of Windows 2000 as well.

Windows DNA Object Model

The Windows DNA object model is analogous to J2EE, and is shown in Figure 2.

In Windows DNA, the presentation tier can include CORBA clients (connecting through a COM-CORBA bridge), ActiveX Controls running within a Web browser, standalone applications, Internet Server API (ISAPI) programs, Active Server Pages (ASP), and static Web pages. Clients use Microsoft's Active Directory to locate middle tier components, and use DCOM to invoke methods on those components. Although it is not shown in Figure 2 (for clarity), messages can also be sent asynchronously using Microsoft Message Queue (MSMQ), COM+ events, or queued component technology.

The Windows DNA business tier contains business and data logic, encapsulated within COM+ components. COM+ components can be written in any language that supports COM+. All invocations to COM+ components are intercepted by the COM+ runtime, and delegated to the components. This gives the COM+ runtime the opportunity to perform middleware operations, such as transactions, security, and object lifecycle management.

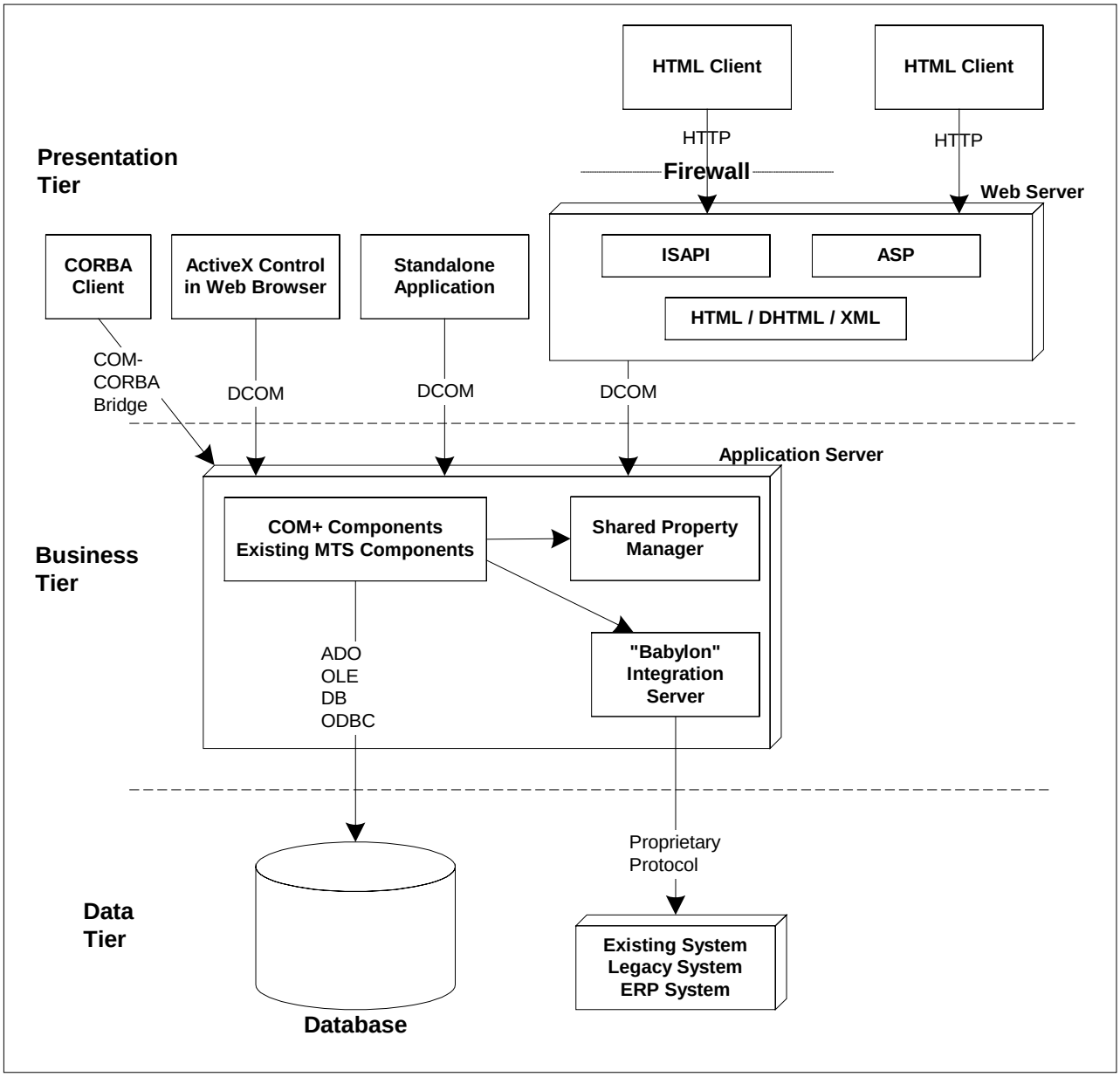

Figure 2: The Windows DNA Object Model

To integrate with databases in Windows DNA, one uses Microsoft's Active Database Objects (ADO) along with OLE/DB and Open Database Connectivity (ODBC). Microsoft's "Babylon" Integration Server provides for connectivity to existing enterprise information systems.

The Windows DNA platform technologies most relevant to our discussion are shown in Table 2.

Windows 2000	The underlying operating system that Windows DNA applications must realistically execute on.
COM+	An architecture for building re-usable, server-side components. COM+ is an evolution of DCOM, MTS, and parts of MSMQ.
Distributed COM (DCOM)	A core technology that promotes interface/implementation separation and language independence and allows for distributed components.
Microsoft Transaction Server (MTS)	A hybrid transaction processing monitor and object request broker that manages server-side components. The Microsoft Distributed Transaction Coordinator (MSDTC) enables two-phase commit transactions to span data sources.
Microsoft Message Queue (MSMQ)	A message-queuing product for asynchronous communications between components.
Microsoft Cluster Server ("Wolfpack")	Allows machines to back each other up to avoid single points of failure.
Network Load-Balancing (formerly Windows Load Balancing in Windows NT 4.0)	Distributes IP traffic; primarily used for load-balancing IP traffic between Web servers.
Component Load-Balancing *(slated for a future Windows DNA release)*	Load-balances between servers running COM+ components.
Microsoft Active Data Objects (ADO) and OLE DB	An API for accessing relational database stores.
Microsoft "Babylon" Integration Server	Used for integration with existing enterprise information systems (such as CICS running on an MVS).
Microsoft Internet Information Server	A Web server that exposes an Internet Server API (ISAPI) for high-performance Web applications.
Active Server Pages (ASP)	Technology that allows Web pages to be dynamically generated. Highly useful for Web designers with scripting knowledge (VBscript or JavaScript).
Microsoft Active Directory	A naming and directory service used to store information about users, computers, and other resources on a network.

Table 2: The Windows DNA technology suite

Technical Comparison of EJB and J2EE Technologies with COM+ and Windows DNA

The remainder of this whitepaper compares EJB and J2EE Technologies with COM+ and Windows DNA in greater detail.

Server-side Components Supported

A robust server-side platform should include support for both business process components as well as data components.

A business process component performs an action on behalf of the client. Examples of business process components include submitting an order, verifying credit card information, or performing a workflow.

By way of comparison, a data component is an in-memory replica of data residing in an underlying storage. Data components encapsulate access to permanent data by wrapping that data in an object, and associating methods with that data. Examples of data components include employees, credit cards, and orders.

The reader should be aware that there is a significant difference between business process components and data components. Business process components are analogous to verbs (they perform an action) whereas data components are analogous to nouns (they are the data that the actions are performed on). For example, logic to transfer funds between bank accounts resides in a *bank teller* business process component, and the two bank account records that the bank teller operates on reside in two *bank account* data components.

The following summarizes the value that data components bring to the table:

Intuitive. Data components are much more intuitive to work with than rectangular database rows. It is much more natural to call *employee.getPassword()* than to deal with a relational result set.

Maps to business problems. Data components offer a higher level of abstraction because they map into real-life business entities. This not only makes development more intuitive, it also maps well into commonly practiced modeling paradigms and enables the use of modeling tools to effectively design enterprise applications.

Object-oriented. One can associate data with data-logic, such as logic to automatically encrypt an employee's password when a client calls *employee.setPassword()*, or logic to automatically decompress an image when a client calls *employee.getPhoto()*.

High re-use. Data components can be re-used in a variety of business processes, some of which may not have been initially envisioned in an application's initial architecture. For example, a single data component that represents a *purchase order* can be used by a wide variety of business processes: *order submission*, *order approval*, *order fulfillment*, and *order reporting*. This re-use is possible because data logic is divorced from business logic.

Underlying schema abstraction. In most systems today, changing a single database column means changing each line of client code that relied on that column, potentially breaking thousands of lines of code. One of the most important values of data components is that they abstract the underlying database schema from business process code. This enables a database administrator to change a relational column name, de-normalize a schema for performance, or migrate to an entirely different database product without modifying a single line of business process code. This is possible because business process components work with data components only and never touch the underlying database directly.

While both these components are powerful, they also require that the developers exercise good design principles. For example, data components should be coarse-grained, representing large chunks of information, such as employees or orders. Data components should not represent small bits of information, such as order line-items or employee addresses. Designing incorrectly may harm the scalability of the overall system.

Component Types Supported in EJB

The EJB specification mandates support for both component types: business process components (session beans) as well as data components (entity beans). This gives developers great flexibility when designing their server-side systems because they can leverage the virtues of both types of components. Since the J2EE test suite tests for support for both component types, every J2EE platform-based product will fully support any developer's object model.

Component Types Supported in COM+

COM+ supports business process components, which are called COM+ components. However, COM+ has no notion of a data component. Because of this, a developer's business process components must operate on rectangular rows, rather than on data components, which is highly non-intuitive. Developers must resort to authoring libraries of components that perform data logic, which is a procedural approach to programming. The result is that COM+ departs from the object-oriented paradigm, and restricts developers from building rich domain object models. By not supporting data components, COM+ deployments do not actualize any of the aforementioned benefits that data components bring to the table, which are necessary for an extensible, robust enterprise system.

Stateless vs. Stateful Business Processes

A *stateful business process* is a business process that spans multiple method calls. A *stateless business process* is a business process that spans a single method call. Stateful business processes require state to be held on the server on behalf of a client, whereas stateless business processes do not.

Recently, there have been many debates regarding stateful vs. stateless programming. Some analysts have advocated that stateful programming should be avoided in the enterprise to enhance scalability and reliability. As we will see, stateful programming can be both well-used, as well as misused. For the times that stateful programming is warranted, however, having support for it in the underlying architecture is a great relief to the application developer.

Stateful programming is well-used when implementing a stateful business process. For example, a shopping cart is well-modeled as a stateful business process component, because a

shopping cart maintains state on behalf of a particular client.

Stateful programming can also be misused, and the prime way to do this is to chain many stateful business process components together. If many stateful components are dependent on one another, and any stateful component fails, then the entire chained conversation has failed. In other words, the weakest link in the chain can break the entire chain. Thus, when using stateful programming, the developer must exercise judgement and avoid chained conversations.

One false claim is that stateful programming lowers the overall scalability of a deployment, because extra resources are consumed. This argument rests on the fact that stateful components cannot be pooled and re-used because they hold state from a particular client. The reality is that pooling stateless components has a single benefit: it lowers the amount of memory being used. No other types of resources are affected, such as database connections or socket connections, as those should be acquired and released just-in-time and should not be part of a component's conversational state. Given that the cost of memory is rarely an issue in a deployment, and that memory prices are at an all-time low, the memory consumption of stateful components is relatively unimportant.

Another argument against stateful programming is that stateful machine crashes result in lost conversations and single points of failure, and so stateful programming should be avoided. This argument is also false; an application developer should use stateful programming if the business problem is naturally stateful. If the developer is concerned about a machine crash resulting in a lost conversation, then the developer should choose a server-side platform that automatically replicates state across machines in a deployment, and will transparently reconstruct the conversation on a different machine upon a crash.

How J2EE Handles Stateful Business Processes

J2EE has implicit support for stateful business processes via stateful session beans. A stateful session bean maintains state on behalf of a particular client, and is tied to that client for the component's lifecycle.

If a system is not deployed with enough memory (perhaps due to unanticipated heavy traffic), the J2EE platform-based product will swap stateful components out to disk, typically by a least-recently-used algorithm, freeing memory for other needs. If a stateful component is needed once again, it is swapped back into memory. The client code is completely unaware that this process has taken place. Since the J2EE platform-based product performs this state management, the application developer does not need to write state management code. This saves the developer time and reduces the application's complexity tremendously, enabling the developer to focus on the business problem at hand, rather than the middleware "plumbing" of the server-side system.

Select vendors implementing the J2EE platform provide a clustered solution, replicating conversational state across J2EE server instances, and transparently failing-over in case of server crashes. Examples of vendors implementing the J2EE platform that provide this quality of service include BEA Systems, Gemstone Systems, IBM Corporation, Persistence Software, Bluestone Software, SilverStream Software, and the Sun-Netscape Alliance.

How Windows DNA Handles Stateful Business Processes

There is very weak support for stateful business processes in Windows DNA. COM+ components cannot hold state beyond a transaction because they are activated and deactivated by the COM+ runtime upon transactional boundaries. To achieve a conversation, developers must manually marshal conversational state into a component upon each transaction, and manually extract that state upon transaction completion.

There are several ways to marshal state into and out of a COM+ component. The strategies include:

Pass the state between the client and the COM+ component upon each transaction. This may result in network bottlenecks if the state is large. It also makes the component's client API unnecessarily complex.

Marshal the state between a database and the COM+ component upon each transaction. This may result in network bottlenecks. It also may result in database bottlenecks. Furthermore, it becomes necessary to write cleanup code that wipes the database of lost conversations.

Marshal the state between the Shared Property Manager (SPM) and the COM+ component upon each transaction. The SPM is an in-memory holder for state that COM+ components use. This approach has many drawbacks. It uses memory (which was the big claim against stateful components), it is a single point of failure (the SPM is not replicated across machines), and the state is not automatically swapped to disk if it gets too large (as stateful session beans are).

Each of these Windows DNA strategies has its own share of limitations that make applications unnecessarily complex, and may harm the scalability of the system. These drawbacks are avoided in high-end J2EE platform-based products. Furthermore, each of these Windows DNA strategies requires the developer to write extra marshaling code. This "plumbing" should be handled for developers by the server-side platform, as it is in J2EE.

Persistence

Persistence is the storage of enterprise data in a durable data store, such as a database. Any serious commerce application requires persistence, just as any serious commerce application requires security and transactions. These are all middleware services that are common to enterprise applications.

In both EJB and COM+ Technologies, middleware services (such as security and transactions) can be specified declaratively, rather than programmatically. This means the developer can *declare* security and transactional needs, rather than *programming* security and transaction code. This provides the following benefits:

1) Declarative programming saves the application developer coding time. It is generally faster to declare middleware needs that the underlying platform will fulfill automatically, rather than to write middleware code explicitly.

2) Declarative programming enables a developer to separate the application from the middleware, which is a very clean and natural separation to make.

3) Declarative programming allows the middleware to be varied without changing source code. This is essential when purchasing components off the shelf, because code might not be available due to intellectual property rights.

Persistence would benefit greatly from the declarative model. Declaring how data should be mapped to an underlying store enables developers to focus on the business problem at hand, rather than muddling application code with persistence logic.

A server-side platform should provide an automated, declarative persistence service to aid the application developer. Such a persistence service should provide adequate functionality to the developer. Things to look for are:

1) Support for persisting complex types of data, such as collections of purchase order line-items.

2) Support for persisting data across multiple data stores, perhaps involving a complex series of joins across several tables.

3) The ability to override the server-side platform's persistence engine and provide custom persistence routines if necessary.

An architecture that provides these features gives developers a higher degree of rapid application development, is powerful enough to address serious business problems, yet is flexible enough to accommodate specialized needs.

How EJB Handles Persistence

In EJB technology, both session beans and entity beans can persist to a database. A session bean can use JDBC or SQL/J to access a database, which is a programmatic approach to persistence. Entity beans are data components, and can be persisted declaratively (*container-managed persistence*), or persisted programmatically (*bean-managed persistence*).

Container-managed persistence is among the most useful features offered by the J2EE platform. The paramount benefit of container-managed persistence is extremely high degree of rapid application development. A typical entity bean that has been persisted manually via JDBC may require 100 lines of code. With container-managed persistence, this can drop to a slim 40 lines of code because the JDBC code has been eliminated[1]. This astounding code size reduction may sound too good to be true, but it is a reality today. Eliminating JDBC code also reduces debugging time significantly; JDBC is very difficult to debug since database access cannot be checked at compile time, only runtime. The end result is developers can spend more time focusing on the application's business logic, rather than data logic, saving time and creating a competitive edge for the corporation.

Another advantage of container-managed persistence is complete database independence. An application vendor, such as SAP, Trilogy, or EC Cubed, is interested in shipping an application that can be deployed on any database that their customers use. It becomes a nightmare for an application vendor to integrate with every vendor's database because each vendor has introduced proprietary extensions to SQL. If an application vendor uses container-

managed persistence, it can ship database-independent application code that can be quickly mapped to any existing customer schema in a declarative fashion. It is even possible to map domain object models to a non-relational data store, such as a flat file or an object database.

Note that there are several caveats to container-managed persistence. For one, the J2EE platform does not specify a standard way to declare persistence needs, because persistence is a very complex subject. Thus, the mechanism of declaring persistence is left to the vendor implementing the J2EE platform, and mappings must be re-specified when deploying an application in a different J2EE platform-based product. But since all mappings are performed at deployment time, rather than at development time, the application code remains portable and unaffected by this mapping step.

When using container-managed persistence, and the domain object model is complex, developers must choose a product that supports complex persistence mappings, since not all vendors' persistence engines are equal. Examples of vendors that implement complex container-managed persistence include IBM Corporation, Persistence Software, The Object People, Secant Technologies, and Thought Inc.

How COM+ handles persistence

COM+ components persist programmatically through ADO or OLE-DB interfaces. OLE-DB can be used to access relational stores, and can be extended to access non-relational data stores as well, such as a file system or Windows registry. ADO is built on top of OLE-DB and provides simple shortcuts for common tasks.

Unfortunately, COM+ does not provide any form of automated persistence support, because there is no notion of a data component in COM+. This has several effects:

Development and maintenance time increases. Since the developer cannot rely on the server-side platform for persistence, the developer must handcraft a custom data mapping layer, which requires significant effort. Furthermore, any feature enhancements or bug fixes generally require longer because of the size and complexity of the code. This increases the amount of time required developing and maintaining a Windows DNA application compared to a J2EE platform-based application.

Deployments are not as adaptable to change. With Windows DNA, developers often write application code that depends upon database-specific SQL extensions. Any applications developed are then bound to a specific database vendor and a specific schema. If the database is ever replaced by a different vendor's offering, or if the database schema ever changes, the data layer will need to be heavily modified. By way of comparison, J2EE platform-based applications that rely on container-managed persistence can be remapped from one vendor's database to another's, or from one database schema to another, without modifying source code. Furthermore, corporations can adapt well to business process changes because the code is less bloated and easier to deal with. This allows for best-of-breed choice of the underlying data storage, and enables a deployment to quickly adapt to changes in the data tier.

Customer support costs increases. If the application is being developed for resale, then the application vendor must rewrite the data access layer for each customer's database vendor and existing schema. This quickly results in a myriad of databases and schemas that must be supported, exponentially increasing customer support costs.

Database Caching Compared

Most every commerce system today contains one or more databases that store persistent data. As the database is usually the bottleneck in a commerce deployment, it is desirable to avoid accessing the database whenever possible. This includes minimizing read operations as well as write operations.

One strategy to minimize the number of database accesses is to cache database data in the middle tier. There are several types of caching in today's server-side platforms:

Result set caching. When an application reads database data, the result set can be cached in the client's address space, allowing the client to scroll through the result set without re-accessing the database. When an application writes database data, all writes to a given result set are deferred until transaction commit time, optimizing on the number of write operations. Result sets are cached for a single transaction only.

Data object caching. A data object caching system caches entire business objects rather than relational rows. When an application reads database data, the data is cached as data objects in the client's address space, allowing the client to access the data objects without re-accessing the database. When an application writes database data, database updates can be deferred until transaction commit time, optimizing on the number of write operations. These cached data objects constitute a long-lived cache, and are accessible across transactions.

Data object caching systems are preferred over result set caching due to the following performance reasons:

Fewer database read operations. With result set caching, the database must be accessed on each transaction, whereas with data object caching, the database does not need to be accessed since data is cached over long periods of time. This results in a tremendous increase in the deployment's scalability.

Fewer database write operations. When a transaction completes and a data object is written to the database, the data object caching system may only make a log entry and defer the write operation until a more convenient time. This also improves the scalability of the commerce system significantly.

Of course, with a data object caching system comes cache consistency issues. If a rogue application updates a database directly without going through the data object caching system, the cache could become inconsistent with the database values. Some data object caching systems provide notification services (such as a trigger) to maintain this synchronization automatically.

Another cache consistency problem arises when a deployment scales to multiple servers and each server has its own cache for the database. If one server's cache is modified, the other servers' caches must be refreshed. A sophisticated server-side platform vendor can solve this issue by providing a *distributed, shared data object cache* that keeps each server's cache automatically synchronized with the others. Such advanced caching algorithms can provide scalability benefits as well. Since the cache is shared, database updates can be deferred for long periods of time, because the cache itself is a consistent, transactional, durable data of record.

J2EE Caching Model

Most J2EE product vendors support result set caching. Select vendors also implement data object caching by using entity beans. An entity bean serves as a natural cache for a database, as it is an in-memory replica of the database values. For example, if a client requested an employee entity bean with an employee ID #256, that employee entity bean could be cached and later re-used if another client requested the same employee. This caching is ideal, because application code is completely unaware that caching is occurring. The application developer does not write to a caching API, because caching is automatically handled by the J2EE product. The application can then be redeployed into a different J2EE platform-based product without changing application code.

Select vendors implementing the J2EE platform also provide a distributed shared object cache for entity beans. Example vendors include Gemstone Systems, Persistence Software, and Object Design Inc. These distributed shared object caching systems provide significant performance advantages to commerce systems, yet enable data integrity to be preserved when deployments are scaled to multiple physical machines.

Windows DNA Caching Model

The Windows DNA architecture provides automatic result set caching for applications. Unfortunately, Windows DNA does not include support for data object caching, nor for distributed shared data object caching. This significantly impairs the scalability of a Windows DNA deployment compared to a J2EE platform-based deployment.

Originally, Microsoft had plans to compensate for this by releasing a long-lived result set caching system, called the In-Memory Database (IMDB), which has been permanently removed from Windows 2000 due to lack of functionality. This illustrates a danger of choosing a proprietary server-side platform, such as Microsoft's Windows DNA: since the application is tightly bound to the underlying vendor implementation, the developer has no choice but to remain with that vendor's implementation, and learn to live with its limitations.

Message-Oriented Middleware (MOM)

Message-oriented middleware support is necessary in large-scale enterprise systems, enabling loose coupling between components, resulting in enhanced scalability. With an asynchronous messaging system, a client can send a message (such as submission of an order) and resume its task immediately, rather than wait for the order to be processed. The order is persisted in a temporary message queue and will eventually be processed when a server is available, perhaps during non-peak time. Furthermore, the client does not have to wait if the server is currently unavailable (perhaps due to a crash). This enhances the overall scalability of the system because more orders can be processed over time. The two most popular messaging models are point-to-point (single message source, single message sink) and publish/subscribe (multiple message sources, multiple message sinks). A message-oriented middleware vendor can provide various qualities of service to messages, including guaranteed delivery in case of network or machine outages.

J2EE MOM Model

The Java Message Service (JMS) API yields both publish/subscribe messaging as well as point-to-point messaging. The JMS API enables a developer to author messaging without hard-coding to a specific vendor's product. It is the responsibility of the application developer to choose a vendor that supports the necessary qualities of service, and this choice does not affect application code. An application developer uses JNDI to locate a JMS driver, and thus JMS applications are not dependent on physical machine names. High-end J2EE platform-based products allow for clustering and load-balancing when delivering messages.

Examples of middleware products that support JMS with guaranteed delivery include BEA's *WebLogic*, Progress Software's *SonicMQ*, IBM's *MQSeries*, and Fiorano Software's *FioranoMQ*.

Windows DNA MOM Model

There are several mechanisms to perform messaging using Windows DNA. Point-to-point messaging is accomplished via queued component technology. With queued components, the developer programs to a component's normal interface, and the method invocation is deferred asynchronously, relieving the application developer from programming to a separate messaging API. This functionality is not currently specified in J2EE. Queued component technology is based upon Microsoft Message Queue (MSMQ), Microsoft's message queuing product, which can also be programmed to directly. If publish/subscribe messaging is desired, the developer can use COM+ Events.

One limitation of Microsoft's messaging architecture is that messages cannot be load-balanced (the subscription is machine-name dependent), although failover is possible through Microsoft Cluster Server ("Wolfpack").

Transactions

Transactions enable developers to perform reliable server-side computing, which is necessary in any mission-critical environment, such as a banking system. Transactions guarantee that a suite of operations execute:

Atomically. A balance transfer between two bank accounts involves a withdrawal from one account and a deposit into another account. Both of these operations must occur together; if any failure occurs, neither operation takes place.

Consistently. A bank account balance can never be negative.

Isolated. Concurrent banking operations do not interfere with each other.

Durably. A crash of the physical bank database is recoverable.

A *distributed transaction* is a transaction that is executed by more than one component, potentially running on different machines. A distributed transaction that spans multiple data sources, such as two different databases, requires a distributed transaction coordination service that implements a two-phase commit protocol.

Today's server-side platforms should give the developer control of transactions both programmatically (hard-coding to a transaction API) and declaratively (setting a transaction

attribute on a component, rather than writing code to an API). It should also be possible to control transactions programmatically from client code (such as a Java servlet beginning and ending a transaction). Finally, there should be a way for application code to register as a transaction participant, alerting the application to the progress of a transaction.

Transactions in J2EE

In J2EE platform-based applications, transactions can be controlled declaratively or programmatically. Declarative transactions can be specified at the EJB component method-level, which means each method in an enterprise bean class can perform transactions differently. Developers can also control transactions programmatically via the Java Transaction API (JTA). Application code can register as a transaction participant by implementing an optional synchronization interface. Thus, developers have great flexibility when designing transactional applications. The J2EE platform-based product can handle transactions automatically, enabling the developer to focus on the business problem at hand. If the developer needs finer grained control of transactions, support is provided there as well.

The underlying transaction service in a J2EE platform-based product is vendor-specific, and does not affect application code. The larger vendors implementing a J2EE platform, such as IBM, BEA, and Oracle, have very mature transaction processing monitors that have been in production for years. These vendors and are repackaging that logic under the hood of a J2EE platform-based product. The result is a very reliable and robust transactional deployment environment for J2EE platform-based applications.

Distributed transaction support is an optional J2EE service. When distributed transactions are required, the application developer should choose a vendor that implements a distributed transaction service. Examples of vendors that implement a distributed transaction service today include IBM, BEA, Inprise, the Sun-Netscape Alliance, GemStone Systems, and Secant Technologies.

Transactions in Windows DNA

In COM+, transactions can also be controlled declaratively and programmatically. Declarative transactions are somewhat limited, however, because different methods within a component interface cannot have different transactional characteristics. The application developer can perform transactions programmatically via OLE Transactions. Application code can register as a transaction participant by authoring a compensating resource manager (CRM). This is a necessary burden with the Microsoft model, as all COM+ components are stateless. Microsoft includes distributed transaction service capabilities in the Microsoft Distributed Transaction Coordinator (MSDTC).

Microsoft also includes support for nested transactions, which enables developers to nest units of work within other units of work. This is an interesting feature that the J2EE architecture does not support. However, the vast majority of today's transactions are not nested, and so most commerce systems do not require this functionality.

Security

A secure deployment provides a mechanism to *authenticate* clients (test whether clients

are who they claim to be) as well as *authorize* clients (tests whether clients have access rights to perform desired operations). Component developers should be able to perform security checks programmatically as well as declaratively. There should also be a mechanism to control whether a component can run with the security level of its client, as well as delegate those security credentials when calling another subsystem.

Security in J2EE

In J2EE, Web clients can be authenticated over a Secure Sockets Layer (SSL) in JSP pages, and servlets. Application code can access credentials in any naming and directory system via JNDI, including Microsoft's *Active Directory* or *Exchange*, Netscape's *Directory Server*, IBM's *Lotus Notes*, or any LDAP server. One limitation of J2EE security today is that portions of authentication code are not portable across J2EE platform-based products. This is expected to be resolved in a future J2EE platform release. Authorization is portable, and can be specified on EJB components either programmatically or declaratively.

J2EE security is based on two separate security models. The Java 2 security model enables the server to control access rights to privileged resources, such as sockets and I/O operations. The Java Authentication and Authorization Service (JAAS) standard extension, slated for a future J2EE platform release, will enable multiple authentication services to be installed into any JAAS compatible application, such as a J2EE platform-based product, and will allow for user-based authorization.

J2EE also enables integration with existing security systems. For example, BEA's *WebLogic* product supports a pluggable security *realm*, an object that performs authentication and authorization. Realms can interoperate with LDAP directory-based security, relational database-based security, Windows security (enabling Windows security tools to interoperate with J2EE deployments), or a user-defined realm. Propagation of security contexts can be performed using IIOP, or in a vendor-specific mechanism that is transparent to application code.

Security in Windows DNA

The Windows DNA security model is quite analogous to J2EE. In Windows DNA, Web clients can be authenticated over a Secure Socket Layer (SSL) in ASPs or ISAPI code. Application code typically accesses credentials in Microsoft's Active Directory. Authorization can be controlled either programmatically or declaratively in COM+ components.

Pluggable security is accomplished in Windows DNA using the Security Service Provider Interface (SSPI). A Simple Negotiation Mechanism selects the proper security service provider. The Windows NT LAN Manager Security Service Provider is one such provider that uses challenge-response algorithm for authentication. Kerberos is used for propagation of security contexts.

Language and Legacy System Support

Most large corporations have existing code written in a variety of languages. A server-side platform should support mechanisms to integrate that code into a deployment to preserve these investments.

Language and Legacy System Support in J2EE

J2EE promotes Java-centric computing, and as such all components deployed into a J2EE container (such as EJB components and servlets) must be written in the Java language. Java is an ideal language for server-side programming, since it is object-oriented, has built-in garbage collection to avoid memory leaks, does not allow for direct pointer access avoiding memory corruption, and is interpreted, all of which make server-side programming safe. Finally, the interpreted nature of Java is a non-issue for performance on the server-side, since server-side bottlenecks are usually network or database related. In the rare event that the speed of Java is an issue, the code can be natively compiled at deployment time with a product such as Tower Technologies' *Tower/J*. This increases scalability yet preserves source code portability.

Despite the advantages of Java technology, Sun Microsystems realizes that existing investments must be preserved, and has supplied several mechanisms to integrate existing code written in other languages into a J2EE deployment:

The Java Native Interface (JNI) can be used to incorporate C++ code into a J2EE deployment.

CORBA can also be used to delegate to code written in a variety of languages, such as C++, COBOL, or Ada. A potential disadvantage to this approach is that CORBA delegation may require network overhead. Fortunately, optimized CORBA products (such as Inprise's *Inprise Application Server*) performs delegation as an Inter-Process Call rather than a Remote Procedure Call, eliminating network bottlenecks. Another potential disadvantage is that CORBA programming requires knowledge of an Interface Definition Language (IDL). Although many CORBA tools will automatically generate IDL on behalf of the developer, we recommend that the developer acquire a thorough understanding of IDL to perform serious CORBA programming and debugging.

The CORBA Component Model (CCM)[2] is a compatible superset of the EJB architecture, and enables true server-side components to be written in any language. At the time of this writing, the Object Management Group (OMG) had just finalized the CCM. Products are expected to appear on the market shortly.

J2EE Connectors (future) will enable standardized integration with existing investments running on legacy, ERP, or other systems. Examples of vendors scheduled to implement the J2EE Connector standard include IBM, which will provide integration with CICS and IMS, SAP which will provide a connector into their R/3 product, and the Sun-Netscape Alliance, which will provide connectors to various existing information systems. Proprietary integration mechanisms exist today.

Language and Legacy System Support in Windows DNA

As COM+ is a binary architecture, it is language-neutral. Code written in almost any language can be theoretically mapped to COM+, although in actuality not all languages are supported by Microsoft's tools.

The tradeoff of language interoperability is added complexity and loss of portability. COM+ achieves language independence by using Microsoft's Interface Definition Language (IDL), which takes a great deal of time to master. Although today's tools hide IDL from developers, to do serious COM+ programming and debugging, one must still have a thorough understanding of IDL.

Other mechanisms of integrating with legacy code include OLE DB (integration with legacy data sources, such as VASM), COM TI (integration with existing Transaction Processing Monitors such as CICS), and MSMQ (integration with existing Message Oriented Middleware products, such as IBM MQSeries).

Load-Balancing

In a typical commerce system, presentation logic executes in a separate physical tier from the business logic, as originally shown in Figures 1 and 2. Separation of tiers allows each tier to scale independently, enables one to secure business process and data logic with a firewall, and enables a Web development team and business process team to each take care of a separate physical part of the enterprise.

To accommodate heavy traffic, one must be able to scale any given tier by adding additional machines to that tier. Adding additional machines requires logic to *load-balance* client requests across machines. There must be logic to load-balance incoming requests to the presentation tier, as well as load-balance requests from the presentation tier to the business tier.

Load-Balancing Support in J2EE

The J2EE platform specification enables a middleware vendor to implement a wide variety of load-balanced solutions. J2EE components automatically benefit from underlying load-balancing services; no change to application source code or application binaries is needed.

J2EE platform-based products that load-balance today include BEA's *WebLogic*, IBM's *WebSphere*, Bluestone's *Sapphire/Web*, Progress' *Apptivity*, and many others. Many of today's vendors implement a wide variety of load-balancing strategies, including round-robin, random, and dynamic (active) load-balancing. The result is a choice of several vendor solutions that offer strong scalability features.

Load-Balancing Support in Windows DNA

Windows DNA contains load-balancing technology to spread IP traffic among Web servers, which is called Network Load-Balancing. This technology is effective at balancing incoming requests to Web server machines.

Microsoft does not currently offer load-balancing technology that directs requests from

the presentation tier to the business tier. Originally, this *component load-balancing* technology was slated for the February 2000 release of Windows DNA, and has been delayed until a future release of Windows 2000 due to lack of functionality. This means that the business tier of a Windows DNA deployment (the tier that contains COM+ components) is currently scalable to a single machine, which can support at most 32 processors. By way of comparison, a J2EE platform-based deployment that load-balances across high-end UNIX or mainframe systems can scale into thousands of processors.

The takeaway point from this discussion is that by choosing a proprietary architecture such as Windows DNA, one is limited by the physical scalability constraints of the software and hardware supported by the proprietary solution. In contrast, an open architecture such as J2EE decouples the application from the underlying operating system and physical hardware, enabling one to choose the most scalable solution for the business problem at hand.

Conclusions

In this whitepaper, we have examined the technical benefits of J2EE and EJB Technologies over Windows DNA and COM+. Windows DNA is currently lacking in critical areas of functionality, particularly in the areas of component types supported, state management, data caching, declarative persistence, and load-balancing. These are necessary features that can be harnessed to create well-designed, maintainable, and scalable solutions. By way of comparison, many of today's J2EE platform-based products provide robust support for each of these features, enabling developers to craft enterprise-class deployments that meet the needs of today's businesses.

References

[1] Source: Code comparison from the book, "Mastering Enterprise JavaBeans, and the Java 2 Platform, Enterprise Edition" by Ed Roman, published by John Wiley & Sons, 1999

[2] For more information about the CCM, visit `http://www.omg.org`