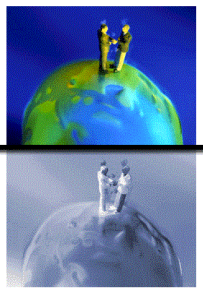

EJB, J2EE, and XML Web Services Expertise

J2EE vs. Microsoft.NET

A comparison of building XML-based web services

By Chad Vawter and Ed Roman
June 2001

Prepared for Sun Microsystems, Inc.

Table of Contents

I. Preface

In this whitepaper, we will make a powerful comparison between the two choices that businesses have for building XML-based web services: the **Java 2 Platform, Enterprise Edition (J2EE)**[1], built by Sun Microsystems and other industry players, and **Microsoft.NET**[2], built by Microsoft Corporation.

Some of the statements we make will offend you, and hopefully more of them will agree with you. So as you read this paper, please remember our three promises:

1. We promise to compare these choices at a logical, neutral, and unbiased level.
2. We promise to tell the tale about how we really do feel about these technologies.
3. We promise to dispel the Fear, Uncertainty, and Doubt (FUD) that exists in the marketplace today.

Although both J2EE and .NET cover a great deal of technologies and standards, we will focus specifically on building server-side systems as web services using these architectures (for example, we will not be mentioning Jini or Office XP). After reading this white paper, you will have a solid understanding of how these architectures compare, and be empowered to make intelligent decisions in new web services initiatives.

The first half of this whitepaper is background information about web services, J2EE, and .NET. If you already understand these technologies, feel free to skip ahead to the 2nd half of the paper, which is the juicy comparison.

II. Introduction

The next generation of distributed computing has arrived. Over the past few years, XML has enabled heterogeneous computing environments to share *information* over the World-Wide Web. It now offers a simplified means by which to share *process* as well. From a technical perspective, the advent of **web services** is not a *revolution* in distributed computing. It is instead a natural *evolution* of XML application from structured representation of information to structured representation of inter-application messaging. The revolution is in the opportunities this evolution affords.

Businesses have been offering products and services on the World-Wide Web for the past few years. Have they not then been offering web services? In what way are *web services* actually new? In an article entitled The Web Services (R)evolution - Applying Web Services to Applications" Graham Glass, the CEO and Chief Architect of The Mind Electric defines a web service as:

> "A collection of functions that are packaged as a single entity and published to the network for use by other programs. Web services are building blocks for creating open distributed systems, and allow companies and individuals to quickly and cheaply make their digital assets available worldwide."[3]

[1] The J2EE homepage: http://java.sun.com/j2ee
2 The Microsoft .NET homepage: http://www.microsoft.com/net/
3 http://www-106.ibm.com/developerworks/webservices/library/ws-peer1.html?dwzone=ws

Prior to the advent of web services, enterprise application integration was very difficult due to differences in programming languages and middleware used within organizations. The chances of any two business systems using the same programming language and the same middleware was slim to none, since there has not been a de-facto winner. These 'component wars' spelled headaches for integration efforts, and resulted in a plethora of custom adapters, one-off integrations, and integration 'middlemen'. In short, interoperability was cumbersome and painful.

With web services, any application can be integrated so long as it is Internet-enabled. The foundation of web services is XML messaging over standard web protocols such as HTTP. This is a very lightweight communication mechanism that any programming language, middleware, or platform can participate in, easing interoperability greatly. These industry standards enjoy widespread industry acceptance, making them very low-risk technologies for corporations to adopt. With web services, you can integrate two businesses, departments, or applications quickly and cost-effectively.

The vision for web services predicts that services will register themselves in public or private business registries. Those web services will fully describe themselves, including interface structure, business requirements, business processes, and terms and conditions for use. Consumers of those services read these descriptions to understand the abilities of those web services. Web services will be smart, in that once a service has been invoked, it will spontaneously invoke other services to accomplish the task and to give users a completely personal, customized experience. In order for these services to dynamically interact, they need to share information about the user's identity, or context information. That context information should only need to be typed in once, and then made available at the user's discretion to selected web services.

Building web services with technologies that have gained the most acceptance

Now that we've seen the general philosophy behind web services, let's look at how to build and use a web service. Web services are in reality simply XML-based *interfaces* to business, application, and system services, and are really old technologies wearing a new hat. The following technologies that have gained the most industry acceptance, and is one possible way to perform web services:

- A provider creates, assembles, and deploys a web service using the programming language, middleware, and platform of the provider's own choice.
- The provider defines the web service in WSDL (the **Web Services Description Language**[4]**)**. A WSDL document describes a web service to others[5].
- The provider registers the service in UDDI **(Universal Description, Discovery, and Integration**[6]**)** registries. UDDI enables developers to publish web services and that enables their software to search for services offered by others.
- A prospective user finds the service by searching a UDDI registry.
- The user's application binds to the web service and invokes the service's operations using SOAP (the **Simple Object Access Protocol**[7]**)**. SOAP offers an XML format for representing

[4] The Web Services Description Language (WSDL) 1.1 specification is available at http://www.w3.org/TR/wsdl
[5] Visit http://www.xmethods.com/ for a listing of some interesting web services, and links to their accompanying WSDL documents.
[6] More information regarding the UDDI initiative is available at http://www.uddi.org/
[7] The SOAP specification is available at http://www.w3.org/TR/SOAP/

parameters and return values over HTTP. It is the communications protocol that all web services use.

Note that the above technologies are only sufficient for simple web services. Extended business exchanges require an agreed-upon structure for business transactions, multi-request transactions, schemas, and document flow. These application requirements often stretch the limits of a purely SOAP based implementation. This is the motivation for **ebXML**, which is a suite of XML specifications and related processes and behavior designed to provide an e-infrastructure for B2B collaboration and integration.

Note that the above approach is but one way of making web services work. There are other choices as well, but we feel that these technologies are the most important and will achieve the widest industry adoption. Because of this, in reality, we really haven't reached complete consensus on building web services, and there are still a lot of issues to be resolved. For example, there is vendor disagreement on SOAP extensions, ebXML, and service flow descriptions. The good news is that:

- For once, all major players, including Sun and Microsoft, generally agree that SOAP, WSDL, and UDDI are good things and that they (or their standard derivatives) will provide a foundation for the future.
- All the vendors are working together to establish web services standards, and a foundation is emerging.

The J2EE and Microsoft.NET approach to Web Services

If you want to build a usable web services system, there is more than meets the eye. Your web services must be reliable, highly available, fault-tolerant, scalable, and must perform at acceptable levels. These needs are no different than the needs of any other enterprise application.

J2EE and .NET are evolutions of existing application server technology used to build such enterprise applications. The earlier versions of these technologies have historically *not* been used to build web services. Now that web services has arrived, both camps are repositioning their solutions as platforms that you can also use to build web services.

The shared vision between both J2EE and .NET is that there is an incredible amount of 'plumbing' that goes into building web services, such as XML interoperability, load-balancing, and transactions. Rather than writing all that plumbing yourself, you can write an application that runs within a container that provides those tricky services for you.

This paradigm allows you to specialize in your proficiencies. If you were a financial services firm, for example, you'd have proficiency in financial services, but likely very little proficiency in web services plumbing compared to a specialist such as Sun, IBM, BEA, Oracle, or Microsoft. By purchasing the container off-the-shelf, you won't need to be an expert at plumbing to build a financial services-based web service. Rather you just need to understand their business problem at hand, and leave the web service plumbing to the container.

With that said, let's take a look at the details of each vision.

III. J2EE

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 industry standard, and is the result of a large industry initiative led by Sun Microsystems.

It's important for you to realize that J2EE is a standard, not a product. You cannot "download" J2EE. Rather you download a set of Adobe Acrobat PDF files which describe agreements between applications and the containers in which they run. So long as both sides obey the J2EE contracts, applications can be deployed in a variety of container environments.

The J2EE camp's goal is to give customers choice of vendor products and tools, and to encourage best-of-breed products to emerge through competition. The only way this would ever happen is if the industry as a whole were bought-into J2EE. To secure buy-in, Sun collaborated with other vendors of eBusiness platforms, such as BEA, IBM, and Oracle, in defining J2EE. Sun then initiated the Java Community Process (JCP) to solicit new ideas to improve J2EE over time. The reason Sun did this is because they had to do so to achieve success--the best way to secure buy-in to an idea is to involve others in defining that idea.

Java: The foundation for J2EE

The J2EE architecture is based on the Java programming language. What's exciting about Java is that it enables organizations to write their code once, and deploy that code onto any platform. The process is as follows:

1. Developers write source code in Java.
2. The Java code is compiled into **bytecode**, which is a cross-platform intermediary, halfway between source code and machine language.
3. When the code is ready to run, the Java Runtime Environment (JRE) interprets this bytecode and executes it at run-time.

J2EE is an application of Java. Your J2EE components are transformed into bytecode and executed by a JRE at runtime. Even the containers are typically written in Java.

J2EE and Web Services

J2EE has historically been an architecture for building server-side deployments in the Java programming language. It can be used to build traditional web sites, software components, or packaged applications. J2EE has recently been extended to include support for building XML-based web services as well. These web services can interoperate with other web services that may or may not have been written to the J2EE standard.

J2EE web services development model is shown in Figure 1.

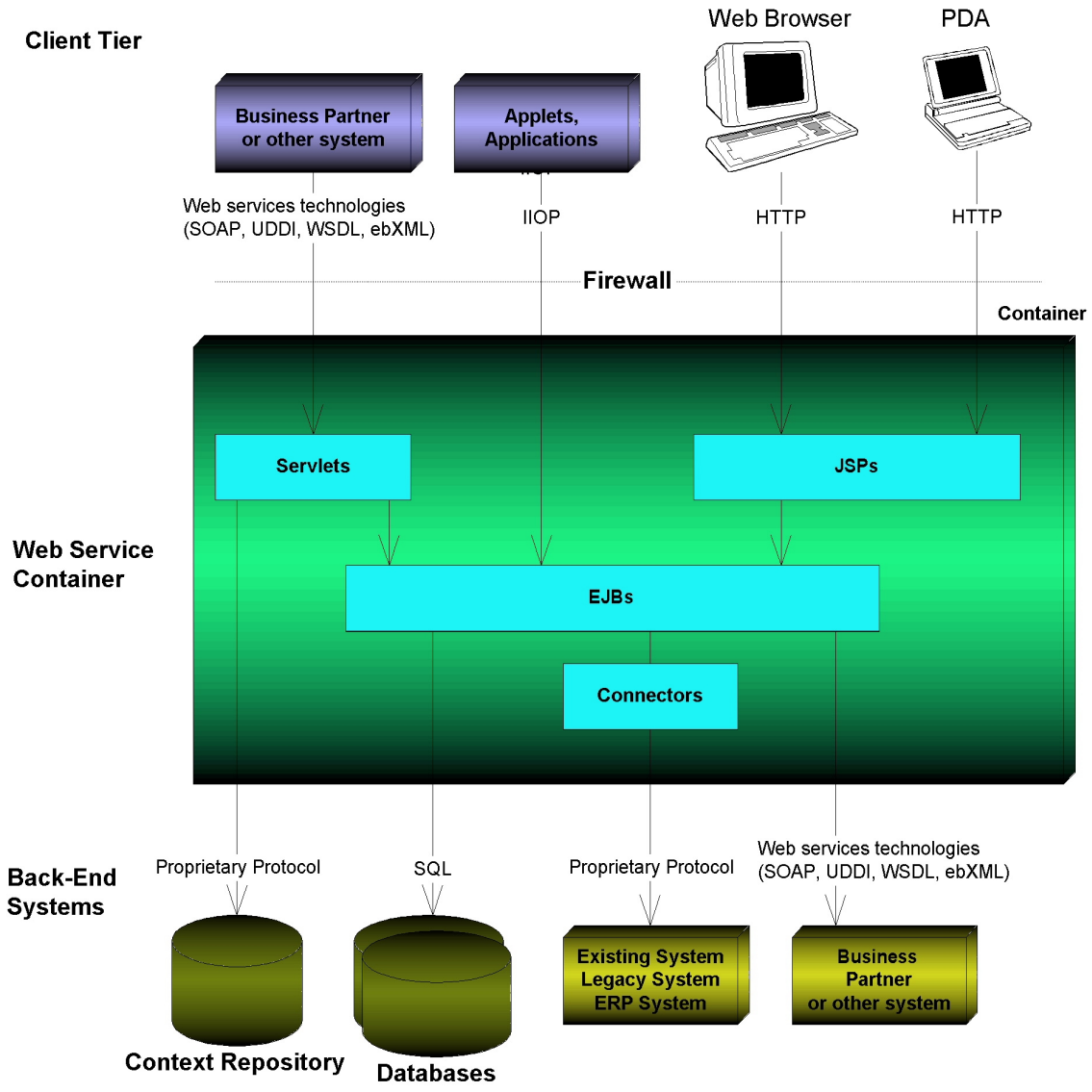

Figure 1 Developing web services with J2EE

Briefly, Figure 1 is explained as follows:

J2EE application is hosted within a container, which provides qualities of service necessary for enterprise applications, such as transactions, security, and persistence services.

The business layer performs business processing and data logic. In large-scale J2EE applications, business logic is built using Enterprise JavaBeans (EJB) components. This layer performs business processing and data logic. It connects to databases using Java Database Connectivity (JDBC) or SQL/J, or existing systems using the Java Connector Architecture (JCA). It can also connect to business partners using web services technologies (SOAP, UDDI, WSDL, ebXML) through the Java APIs for XML (the JAX APIs).

Business partners can connect with J2EE applications through web services technologies (SOAP, UDDI, WSDL, ebXML). A servlet, which is a request/response oriented Java object, can accept web service requests from business partners. The servlet uses the JAX APIs to perform web services

operations. Shared context services will be standardized in the future through shared context standards that will be included with J2EE.

Traditional 'thick' clients such as applets or applications connect directly to the EJB layer through the Internet Inter-ORB Protocol (IIOP) rather than web services, since generally the thick clients are written by the same organization that authored J2EE application, and therefore there is no need for XML-based web service collaboration.

Web browsers and wireless devices connect to JavaServer Pages (JSPs) which render user interfaces in HTML, XHTML, or WML.

Reference Implementation

In addition to the specifications, Sun also ships a reference implementation of J2EE. Developers write applications to this to ensure portability of their components. This implementation should not be used for production, but rather just for testing purposes.

Additional Services

All vendors that offer J2EE platforms provide additional features not found in the standard. Some of them impact portability, such as extended EAI functionality, E-Commerce components, or advanced B2B integration. Other features, such as load-balancing, transparent fail-over, and caching, do not affect portability of application code, because they are *implicit* services which are provided behind-the-scenes by the container.

IV. Microsoft's .NET Platform

Microsoft.NET[8] is product suite that enables organizations to build smart, enterprise-class web services. Note the important difference: .NET is a product strategy, whereas J2EE is a standard to which products are written.

Microsoft.NET is largely a rewrite of Windows DNA, which was Microsoft's previous platform for developing enterprise applications. Windows DNA includes many proven technologies that are in production today, including Microsoft Transaction Server (MTS) and COM+, Microsoft Message Queue (MSMQ), and the Microsoft SQL Server database. The new .NET Framework replaces these technologies, and includes a web services layer as well as improved language support.

The developer model for building web services with Microsoft.NET is shown in Figure 4.

8 For further information regarding Microsoft .NET, please see the .NET homepage at http://www.microsoft.com/net/

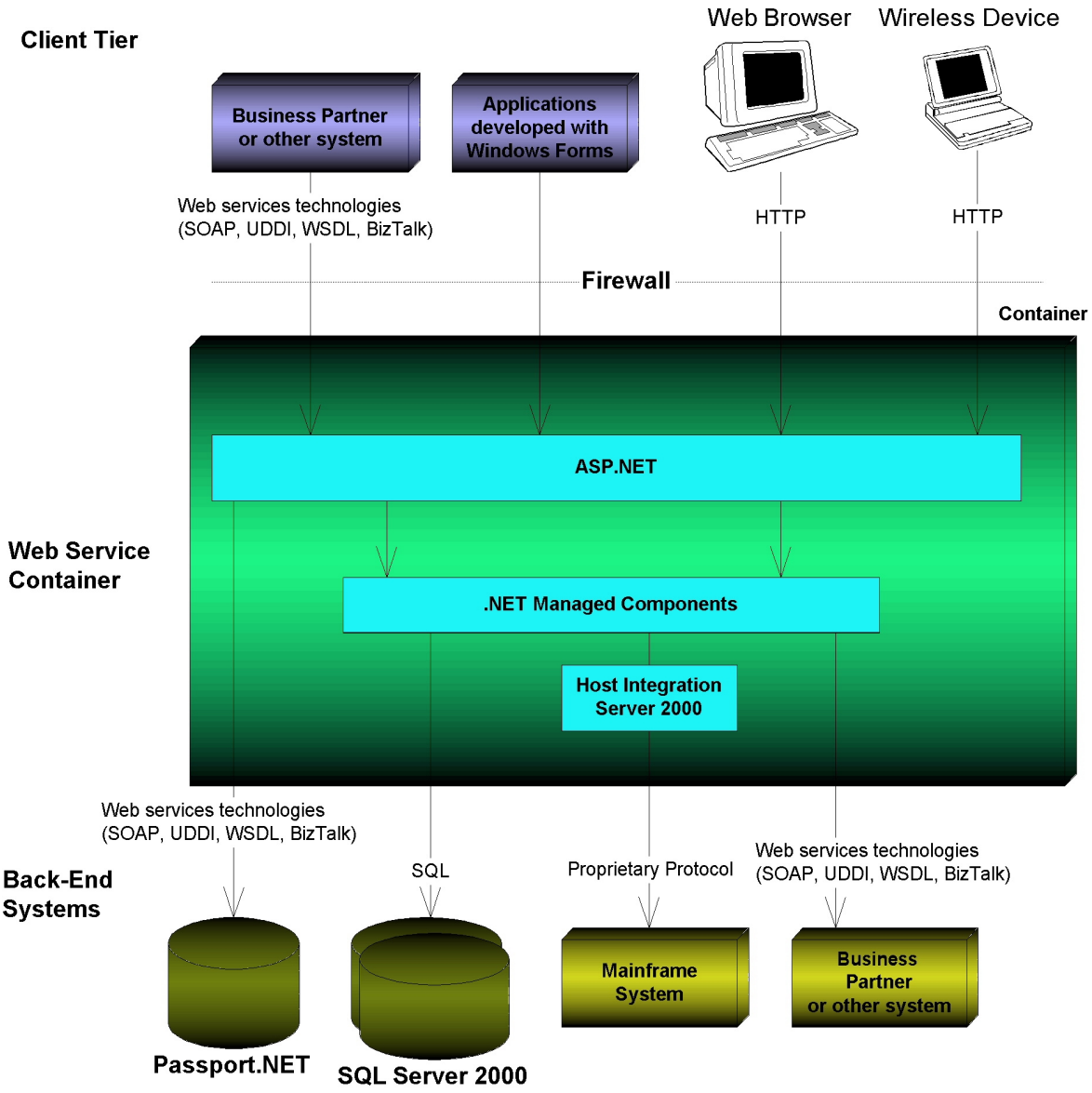

Figure 4 Developing web services with Microsoft.NET

Briefly, Figure 4 is explained as follows:

The .NET application is hosted within a container, which provides qualities of service necessary for enterprise applications, such as transactions, security, and messaging services.

The business layer of the .NET application is built using .NET managed components. This layer performs business processing and data logic. It connects to databases using Active Data Objects (ADO.NET) and existing systems using services provided by Microsoft Host Integration Server 2000, such as the COM Transaction Integrator (COM TI). It can also connect to business partners using web services technologies (SOAP, UDDI, WSDL).

Business partners can connect with the .NET application through web services technologies (SOAP, UDDI, WSDL, BizTalk).

Traditional 'thick' clients, web browsers, wireless devices connect to Active Server Pages (ASP.NET) which render user interfaces in HTML, XHTML, or WML. Heavyweight user interfaces are built using Windows Forms.

The .NET Framework

Microsoft.NET offers language-independence and language-interoperability. This is one of the most intriguing and fundamental aspects of the .NET platform. A single .NET component can be written, for example, partially in VB.NET, the .NET version of Visual Basic, and C#, Microsoft's new object-oriented programming language.

How does this work? First, source code is translated into **Microsoft Intermediate Language**, sometimes abbreviated **MSIL**, sometimes **IL**. This IL code is language-neutral, and is analogous to Java bytecode.

The IL code then needs to be interpreted and translated into a native executable. The .NET Framework includes the **Common Language Runtime (CLR)**, analogous to the Java Runtime Environment (JRE), which achieves this goal. The CLR is Microsoft's intermediary between .NET developers' source code and the underlying hardware, and all .NET code ultimately runs within the CLR.

This CLR provides many exciting features not available in earlier versions of Windows DNA, such as automatic garbage collection, exception handling, cross-language inheritance, debugging, and "side-by-side" execution of different versions of the same .NET component.

.NET Servers

The .NET platform includes the following .NET Enterprise Servers. Many of these are repackagings of existing products under a common marketing term:

SQL Server 2000 is Microsoft's relational database.

Exchange 2000 Server is a messaging and collaboration platform useful in developing and running core business services and is now tightly integrated with Windows 2000.

Commerce Server 2000 offers you quicker and less complicated development and deployment of customizable online e-commerce solutions.

Application Center Server 2000. Application Center Server 2000 lets you manage clustered servers.

Host Integration Server 2000. Host Integration Server 2000 gives you access to selected legacy systems running on other platforms (primarily IBM-based).

Internet Security and Acceleration (ISA) Server 2000 offers firewall and Web caching capabilities.

BizTalk Server 2000 is Microsoft's XML-based collaborative e-business solution for integrating applications, trading partners and business processes via the Internet.

Other .NET Products and Services

A new version of Visual Studio, another of Microsoft's flagship products, has been released as **Visual Studio.NET**, an integrated development environment for the .NET platform. All languages supported by Visual Studio prior to the release of .NET (except for Java) are still supported by Visual Studio.

Visual Studio.NET also provides support for Microsoft's new **C#** language which is semantically equivalent to Java, with just a few minor syntactical differences.

Hailstorm[9] is Microsoft's portfolio of building block web services. Microsoft and possibly Microsoft partners will host the Hailstorm web services. Some Hailstorm web services will be available on a subscription basis, and others will be free. For example, Microsoft's Passport service is now freely available as a web service-oriented universal identification mechanism.

Web services such as Microsoft's Passport offer a *shared context* in that many other web services can depend upon Passport for identity and security credentials. User's need not have their contextual information spread across and possibly duplicated across multiple web sites. "Islands" of information can now be located in a single repository and then shared by user-centric web services regardless of the invoking client device.

Privacy of information is focal point in Hailstorm web servicing. In fact, users of Hailstorm web services are empowered to manage the availability of their personal information through "affirmative consent".

Understanding J2EE and .NET by analogy

To help you understand both models, we offer analogies between J2EE and .NET technologies in Table 1. This table only showcases the similarities--we will get to the differences in a few moments.

9 Information regarding Hailstorm is available at http://www.microsoft.com/net/hailstorm.asp

Feature	J2EE	.NET
Type of technology	Standard	Product
Middleware Vendors	30+	Microsoft
Interpreter	JRE	CLR
Dynamic Web Pages	JSP	ASP.NET
Middle-Tier Components	EJB	.NET Managed Components
Database access	JDBC, SQL/J	ADO.NET
SOAP, WSDL, UDDI	Yes	Yes
Implicit middleware (load-balancing, etc)	Yes	Yes

Table 1 Analogies between J2EE and .NET

V. Comparative Analysis

Time-to-Market Features

When developing a commerce solution in today's marketplace, a few months of time is an eternity. Missing a small window of opportunity is the difference between a corporation that is first to market, and a corporation that must play catch-up for years.

One way to speed time to market is to choose a Web services platform that allows rapid application development. This enables developers to write and maintain code quickly, lowering development time.

Both Sun J2EE and Microsoft .NET provide runtime mechanisms that insulate software developers from particular dependencies. In addition to a web service-oriented XML layer of indirection between platforms, languages, and enterprise architectures, Sun J2EE and .NET offer language-level intermediation via the Java Runtime Environment (JRE) and the Common Language Runtime (CLR) respectively.

J2EE offers several features that accelerate time-to-market which are not found in .NET. For example, state management services enable developers to write less code and not worry about managing state, resulting in a higher degree of rapid application development. State management services enable you to build components that hold state. Persistence services (entity beans) enable developers to write applications without coding data access logic, resulting in leaner, database-independent applications that are easier to build and maintain. Programmatic transactions allow you to have greater transactional control. And custom tags are extremely powerful, and empower developers and web designers to easily collaborate.

In addition to these features that enable rapid application development, there are several features that specific vendors offer which aid time-to-market, such as business process management, E-Commerce components, XML-based legacy integration, and enhanced B2B collaboration. A warning: Customers who elect to leverage these features will sacrifice portability. This is a consequence of J2EE architecture

not being all things to all people. However, it should be noted that even after taking advantage of such proprietary features, there would still be a great deal more portability in the end solution than the .NET alternative.

Microsoft.NET offers a variety of time-to-market features not found in J2EE as well. Most notably, ASP.NET is independent of client device, and allows for user interfaces to be rendered to alternative user interfaces without rewriting code. Microsoft also offers Queued Components which are superior to MessageDriven Beans. It should be noted here that Microsoft has tried to simplify server-side programming greatly by removing support for features found in traditional enterprise applications, such as stateful servers and simple transactions. If developers need to go outside this box, their code must be made to be non-managed and reside outside the .NET Framework rather than take advantage of it. Microsoft also provides business process management and E-Commerce capabilities, which are available in some J2EE implementations but not all.

In conclusion, we feel the ability to achieve rapid application development offered by both J2EE and .NET is definitely not equal. It is, however, comparable. The feature differences are minor and it is very difficult to make a compelling argument either way. We do not recommend organizations make their platform decision based upon them. There are larger business issues at hand that dictate the platform choice.

Single-Vendor Solution

When building web services, in general you should always prefer to have a single-vendor solution. A single vendor solution is usually more reliable, interoperable, and less error-prone than a two-vendor bridged solution.

One of J2EE's strengths is that it has spawned a wide variety of tools, products, and applications in the marketplace, which provide more functionality in total than any one vendor could ever provide. However, this strength is also a weakness. J2EE tools are often-times not interoperable, due to imperfections in portability. This limits your ability to mix and match tools without substantial low-level hacking. With lower-end J2EE implementations, you need to mix and match to get a complete solution, and this is the tradeoff when choosing a less complete package. Larger vendors, such as IBM, Oracle, BEA, and iPlanet, each offer a complete web services solution.

.NET provides a fairly complete solution from a single vendor--Microsoft. This solution may lack some of the higher end features that J2EE solutions offer, but in general, the complete web services vision that Microsoft will be providing is equal in scope to that of a larger J2EE vendor.

Another way to look at a single-vendor solution is from a legacy perspective. Many legacy systems are written by J2EE vendors, such as IBM or BEA. J2EE offers a single-vendor solution from the legacy integration perspective, since you can re-use existing relationships with those vendors. A J2EE solution would therefore be a single-vendor solution, since you can stay with that legacy system vendor rather than inject a new vendor such as Microsoft. For users with existing Microsoft-based systems, the reverse argument applies.

Support for Existing Systems

Most large corporations have existing code written in a variety of languages, and have a number of legacy systems, such as CICS/COBOL, C++, SAP R/3, and Siebel. It is vital that corporations be given an efficient, rapid path to preserve and reuse these investments. After all, it is likely that businesses will have neither the funds nor the time to reinvent all existing systems. This legacy integration often is one of the most challenging (if not the most challenging) tasks to overcome when building a web service.

There are several ways to achieve legacy integration using J2EE, including

- The Java Message Service (JMS) to integrate with existing messaging systems
- Web services to integrate with any system
- CORBA for interfacing with code written in other languages that may exist on remote machines.
- JNI for loading native libraries and calling them locally.

But by far, the most important part of the J2EE vision for integration is the J2EE Connector Architecture (JCA). The JCA is a specification for plugging in *resource adapters* that understand how to communicate with existing systems, such as SAP R/3, CICS/COBOL, Siebel, and so-on. If such adapters are not available, you can write your own adapter. These adapters are reusable in any container that supports the JCA. The major vendors of existing systems are bought into the JCA.

.NET also offers legacy integration through the Host Integration Server 2000. COM Transaction Integrator (COM TI) can be used for collaborating transactions across mainframe systems. Microsoft Message Queue (MSMQ) can integrate with legacy systems built using IBM MQSeries. Finally, BizTalk Server 2000 can be used to integrate with systems based on B2B protocols, such as Electronic Data Interchange (EDI) (the reader should note, however, that BizTalk does not serve as an access point to a proprietary network on which EDI takes place).

In conclusion, we believe that the legacy integration features offered by J2EE are superior to those offered by .NET. The JCA market is producing a marketplace of adapters that will greatly ease enterprise application integration. Integration with packaged applications and legacy systems will become much easier--imagine integrating with a system such as Siebel, Oracle, or SAP without ever leaving the Java programming environment. There is no analog to this in the Microsoft domain; rather, there is limited connectivity to select systems provided off-the-shelf through the Host Integration Server.

Market Perception

When comparing two platforms, your first instinct is probably to compare the technologies. The unfortunate reality is that good technology rarely succeeds in the marketplace because it's good technology. Usually it's the technology with the best marketing that wins.

J2EE is an extremely well-marketed platform because it is being marketed by an entire industry of 50+ assorted vendors. This network of interdependent businesses form a virtual marketing machine, and the result is a fantastic market perception for J2EE.

.NET's marketing strength stems from the fact that Microsoft knows how to market a platform. They have put their "A" team on marketing .NET, as is apparent in the industry hype surrounding the platform. This is a powerful force not to be reckoned with. Microsoft's advantage is also that it announced it's web services strategy before the J2EE players, which gave it market perception. So in the marketing front, we give the nod to Microsoft for doing the best job in hyping their platform--so far.

Maturity of Platform

Organizations that adopt a web services platform must consider the maturity of the underlying platform. A less mature, earlier-generation platform is more prone to errors and problems.

J2EE is a veneer atop existing J2EE solutions. J2EE deployments are alive and healthy, running a variety of mission-critical business problems today. However, when looking past the surface, it should be noted that there are some identifiable areas of risk where J2EE lacks maturity:

- The automatic persistence provided EJB is still immature.
- The Java Connector Architecture (JCA) is new.
- All web service support is new.

For Microsoft.NET, the story is a bit different. Some of .NET is based on Windows DNA, which also runs a variety of mission-critical web sites today and enjoys success. However:

- With the new CLR, a good portion of the underlying .NET platform has been substantially rewritten. Indeed, the platform itself is currently only available in a beta version.
- C# is new.
- All web service support is new.

In conclusion, we must find that J2EE is the more mature platform. It is true that certain new features in J2EE are new and risky. However, the very underlying fabric of .NET, is an overhauled rewrite, and the entire C# language is brand new. This represents enormous risk compared to the new J2EE features. This best thing about .NET is that it removes the dependency on the COM Registry -- .NET will do away with DLL Hell. But the worst thing about .NET is that it tosses out the existing infrastructure. We recommend that the risk averse take a 'wait and see' approach with first-generation software such as this.

Language Support

J2EE promotes Java-centric computing, and as such all components deployed into a J2EE deployment (such as EJB components and servlets) must be written in the Java language. To use J2EE, you must commit to coding at least some of your eBusiness systems using the Java programming language. Other languages can be bridged into a J2EE solution through web services, CORBA, JNI, or the JCA, as previously mentioned. However, these languages cannot be intermixed with Java code. In theory, JVM bytecode is language-neutral, however in practice, this bytecode is only used with Java.

By way of comparison, .NET supports development in any language that Microsoft's tools support due to the new CLR. With the exception of Java, all major languages will be supported. Microsoft has also recently introduced its new C# language which is equivalent (with the exception of portability) to Java and is also available as a programming language within the Visual Studio.NET environment. All languages supported by the CLR are interoperable in that all such languages, once translated to IL, are now effectively a “common” language. A single .NET component can therefore be written in several languages.

The multiple language support that Microsoft has introduced with the CLR is an exciting innovation for businesses. It is clearly a feature advantage that .NET has over J2EE. But is the CLR a business advantage for you? This is a more interesting discussion. We are a bit concerned that the CLR may represent a poor design choice for you if more than one language is used. This is for the following reasons:

> **Risk.** Many existing systems are internally convoluted. Disrupting such existing systems is a risky proposition, since knowledgeable engineers, original source code, and a general understanding of the existing system are often-times unavailable. The old adage, "if it ain't broke, don't fix it" applies here.

Maintainability. We speculate that a combination of languages running in the CLR may lead to a mess of combination spaghetti code that is very difficult to maintain. If you have an application written in multiple languages, then to fully develop, debug, maintain, and understand that application, you will need experts in different languages. The need to maintain code written in several languages equates to an increase in developer training expenditures which contributes further to an increased total cost of ownership.

Knowledge building. With combination language code, your developers are unable to share best practices. While individual productivity may increase, communication breaks down, and team productivity decreases.

Skills transfer. While developers using different languages may have very quickly coded a .NET system using VB.NET and C#, what happens if the new C# developers leave your organization? You have two choices. Train your VB.NET developers to understand and write code with C#, or hire other C# developers who know nothing about your code base. The resulting lack in productivity equates to a reduced time to market and a higher total cost of ownership.

In most cases, we feel that it's much better design to standardize on a single language, and to treat legacy systems as legacy systems and integrate with them by calling them through legacy APIs, which can be achieved using either J2EE or .NET.

We do feel the CLR still adds significant value. The value is that a new eBusiness application can be written in a single language of choice other than Java. This is useful for organizations that are not ready to embrace Java. However, again we must provide words of caution.

- It will not be seamless to transition existing developers from their familiar language into productive .NET developers. Procedural languages such as COBOL and VB are being rewritten for .NET to be object-oriented. Teaching developers object-oriented programming is much more of a stepping stone than understanding syntactical rules.
- Languages such as COBOL or VB were never intended to be object-oriented. Legacy code will not seamlessly transition into .NET. The resulting code is forever bound to .NET and can never be taken from its .NET home.
- We question the general wisdom in reinvesting in outdated technologies, such as COBOL, with new eBusiness or web services initiatives.

In summary, there are pros and cons to both approaches to language support. Use the approach that best suits your business needs, but at least be aware of the consequences of your decision.

Migration from Previous Platform

For organizations who have an existing deployment using either J2EE-based technologies or Windows DNA-based technologies, an interesting discussion is the ease of migration from the previous platform to the new platform.

J2EE does not impose many migration problems. As previously mentioned, the Java Connector Architecture (JCA) as well as the web services support in J2EE is brand new and will require new code, but those are minor overall.

Although Microsoft.NET is based on MTS and COM+, we are concerned that the migration to .NET will be taxing compared to J2EE. First off, .NET is based on the “managed code” framework, which steals a

lot of ideas from COM+ and MTS, but it's still an entirely new infrastructure based on an entirely new code base – CLR. Taking advantage of the most valuable aspects of the CLR impose one-time frictions.

For example to accommodate a Common Type System (CTS) which standardizes on data types used between languages, the original Visual Basic data types have been dismissed. Consequently, code dependent upon those original Visual Basic data types will break, and there is currently no migration tool.

Another example is the COM+ migration path. In .NET terminology, code that runs within the CLR is referred to as **managed code**, while code running outside the CLR is called **unmanaged code**. If you're a COM+ developer and want to take advantage of the new CLR, then you have two options for migration:

> **Rewrite existing code as CLR code.** COM+ code needs to be rewritten to accommodate the CLR's automatic garbage collection mechanism and its deprecation of pointers. Dependencies also need to be removed to the COM registry.
>
> **Keep your existing code as unmanaged.** To collaborate between managed and unmanaged code, special measures must be taken.

So as you can see, migration is not free. But to Microsoft's credit, we do understand that with the innovation of the CLR, that this is a necessary step for their customers to evolve into their new platform, and with such a radical change nothing less could be expected. However, we feel obligated to warn users that the migration path will not be easy compared to J2EE migration path, as some might have you believe. Consider these statements from a recent Gartner report:

> "'This is fundamentally a brand new platform,' said Gartner Analyst Mark Driver, comparing the migration to .NET as more drastic than the switch from MS-DOS to Windows. 'This is the tiger changing its stripes...the migration to .NET will be a difficult one for IT departments because it represents such a major shift from the current Microsoft platforms. For instance, developers will have to rewrite as much as 60 percent of the code for some existing Windows applications if they want them to take advantage of Microsoft's .NET platform, Gartner analysts predicted. That's a frightening prospect for companies who are currently switching to Windows 2000, or for those who still run Windows 98 and NT."

Portability

A key difference between J2EE and .NET is that J2EE is platform-agnostic, running on a variety of hardware and operating systems, such as Win32, UNIX, and Mainframe systems. This portability is an absolute reality today because the Java Runtime Environment (JRE), on which J2EE is based, is available on any platform.

There is a second, more debatable aspect of portability as well. J2EE is a standard, and so it supports a variety of implementations, such as BEA, IBM, and Sun. The danger in an open standard such as J2EE is that if vendors are not held strictly to the standard, application portability is sacrificed. CORBA, for example, did not have any way to enforce that CORBA middleware did indeed comply with the standard, and thus there were numerous problems with portability. In the early days of J2EE there were the same problems.

To help with the situation, Sun has built a J2EE compatibility test suite, which ensures that J2EE platforms comply with the standards. This test suite is critical because it ensures portability of

applications. At the time of this writing, there were 18 application server vendors certified as J2EE-compatible. There are a myriad of other vendors as well that are not certified[10].

Our opinion is that in reality, J2EE portability will never be completely free. It is ridiculous to think that complex enterprise applications can be deployed from one environment to the next without any effort, because in practice, organizations must occasionally take advantage of vendor-specific features to achieve real-world systems. However--and this is important--portability is exponentially cheaper and easier with J2EE and the compatibility test suite than with proprietary solutions, and that is a fact we stand behind through years of consulting with customers using a variety of J2EE solutions. Over time, as the J2EE compatibility test suite becomes more and more robust, portability will become even easier.

By way of comparison, .NET only runs on Windows, its supported hardware, and the .NET environment. There is no portability at all. It should be noted that there have been hints that additional implementations of .NET will be available for other platforms. However, a question remains – how much of the complete .NET framework will be (or even can be) supplied on other platforms? History has taught us to be skeptical of Microsoft's claims of multiple platform support. Microsoft ported COM to other platforms, but never ported the additional services associated with COM that were necessary to make COM useful. We find it hard to believe that .NET portability will ever become a reality given Microsoft's historically monopolistic stance.

So how important is portability to you? This is the key question businesses must ask themselves. When evaluating the importance of portability, there are three scenarios worth considering.

- If your firm is selling software to other businesses, or if you are a consulting company, and your customers are on a variety of platforms, we recommend specializing in J2EE architecture. Unless you can guarantee that every one of your customers will accept a Windows/.NET solution, you are restricting your salespeople from major accounts that may have solutions deployed on UNIX or mainframes. This is rarely acceptable at most ISVs or consulting firms.

- If, on the other hand, your customers are on the Windows platform, then either J2EE or .NET will suffice, since they both run on Windows. You should then ask your sales force and consultants what middleware your customers are using on that platform, and make your architecture decision from there. It's important to really be proactive and get this information--the more data you have, the better.

- If you host your own solutions, then you control the deployment environment. That enables you to pick J2EE as well as .NET. If you are willing to standardize on the Win32 platform, and live with the advantages and disadvantages of that platform exclusively, then platform neutrality is irrelevant, and you should consider other factors when deciding on J2EE or .NET. But we offer a word of caution: you can never predict the future. Business goals might change, new vendors might be introduced into the picture, and mergers and acquisitions might happen. All of these may result in a heterogeneous deployment environment. Your applications will not be portable to those platforms in this scenario.

We offer a final opinion to end the debate about just how important is portability. Although both J2EE and .NET will each have their audiences, think about what platform you would choose if you were an ISVs or a consulting company. Most likely you'd have customers on a variety of platforms, and therefore, you'd probably adopt the architecture behind J2EE because you don't want to lock yourself out of deals.

[10] Check out http://www.flashline.com/for a matrix of J2EE offerings, and http://www.theserverside.com/ for reviews of these vendor products.

Indeed, we are noticing today that an incredible number of consulting firms and ISVs are standing behind J2EE, such as Vignette, Broadvision, Chordiant, Kana, NaviSys, and Versata.

What does this mean? It's not about the platform, it's about the applications. As time goes on, customers will likely choose the solution that not only provides the web services infrastructure, but the most consulting support and ISV applications as well.

This makes the future of J2EE very bright in our minds, and is one of *the* critical differentiators between J2EE and .NET.

Web Services Support

The future of eBusiness collaboration is undoubtedly web services. For organizations that are pursuing a web services strategy, or are preparing for the future of web services, their underlying eBusiness architecture must have strong web services support.

Today, J2EE supports web services through the Java API for XML Parsing (JAXP). This API allows developers to perform any web service operation today through manually parsing XML documents. For example, you can use JAXP to perform operations with SOAP, UDDI, WSDL, and ebXML.

Additional APIs are also under development. These are convenience APIs to help developers perform web services operations more rapidly, such as connecting to business registries, transforming XML-to-Java and Java-to-XML, parsing WSDL documents, and performing messaging such as with ebXML.

A variety of J2EE-compatible 3rd party tools are available today that enable rapid development of web services. There are at least sixteen SOAP implementations that support Java. Almost all of these implementations are built on J2EE (servlets or JSP). There are only five UDDI API implementations available, and four of them support Java (IBM UDDI4J, Bowstreet jUDDI, The Mind Electric GLUE, and Idoox WASP). Third-party software vendors such as Tradia (www.tradia.com), CapeClear (www.capeclear.com) and The Mind Electric (www.themindelectric.com) also offer tools for creating web services.

The preview release of Microsoft.NET also enables organizations to build web services. The tools that ship with Microsoft.NET also offer rapid application development of web services, with automatic generation of web service wrappers to existing systems. You can perform operations using SOAP, UDDI, and SDL (the precursor to WSDL). Visual Studio.NET provides wizards that generate web services.

Our conclusions from our web services comparison are as follows.

With J2EE, you can develop and deploy web services today using JAXP. However, this is not the ideal way to build web services, since it requires much manual intervention. An alternative is for organizations to leverage 3rd party libraries to accelerate their development. In the future these libraries will be standardized through the JAX APIs. For now, if you develop web services rapidly, you'll need to bundle these libraries with your application.

With .NET, you can develop web services today using the partial release of .NET. However, since this is only a beta implementation, it does not represent a realistic deployment platform. Another issue with .NET is that it does not support true web services because of a lack of support for ebXML. ebXML is a very important standard for eBusiness collaboration, and is experiencing broad adoption from around the

world. Thousands of vendor and non-vendor companies, government institutions, academic and research institutions, trade groups, standards bodies, and other organizations have joined the ebXML community. This includes HL7 (Health Care), OTA (Open Travel Alliance), RosettaNet, OAG (Open Applications Group), GCI (Global Commerce Initiative), and DISA (Data Interchange Standards Association). Undoubtedly, ebXML is going to be an important force in web services, and we hope that Microsoft chooses to embrace it. Microsoft is still clinging to their BizTalk proprietary framework which has proprietary SOAP extensions. This evidence makes us question Microsoft's true commitment to open and interoperable web services.

Tools

The Sun J2EE Product Portfolio includes Forte, a modular and extensible Java-based IDE that pre-dates both Sun J2EE and .NET. Developers who prefer other IDEs for Java development are free to use WebGain's Visual Café, IBM's VisualAge for Java, Borland's JBuilder, and more. Numerous 3rd party tools and open source-code products are available.

Microsoft has always been a strong tools vendor, and that has not changed. As part of its launch of .NET, Microsoft released a beta version of the Visual Studio.NET integrated development environment. Visual Studio.NET supports all languages supported by earlier releases of Visual Studio - with the notable exception of Java. In its place, the IDE supports C#, Microsoft's new object-oriented programming language, which bears a remarkable resemblance to Java. Visual Studio.NET has some interesting productivity features including Web Forms, a web-based version of Win Forms, .NET's GUI component set. Visual Studio.NET enables developers to take advantage of .NET's support for cross-language inheritance.

Our conclusion is that Microsoft has the clear win when it comes to tools. While the functionality of the toolset provided by J2EE community as a whole supercedes the functionality of the tools provided by Microsoft, these tools are not 100% interoperable, because they do not originate from a single vendor. Much more low-level hacking is required to achieve business goals when working with a mixed toolkit, and no single tool is the clear choice, nor does any single tool compare with what Microsoft offers in Visual Studio.NET. Microsoft's single-vendor integration, the ease-of-use, and the super-cool wizards are awesome to have when building web services.

Shared Context

A key element of smart web services is *shared context*. To understand shared context, think about how many usernames, passwords, credit card information, and so-on that you need to remember and re-type in every time you visit a web site. The vision for shared context is that you type this information in once, and that information is then accessible to all web services that you choose to give access to that information. The information is under your control, rather than the control of the web services, and is protected using security rules that you define.

The Sun J2EE vision for shared context is a decentralized, distributed suite of shared context services that live on the Internet. Each user might have a list of one or more of their preferred shared context services, with each repository storing select information about that user. You would point the web service to your preferred shared context service when you connect.

Today, developers are empowered to create shared context services by writing a servlet that exposes itself as a web service using JAXP, and by using JDBC to access a relational storage.

In the future, Sun J2EE will include standards to access shared context services. This will enable the context repositories to act as web services using a standard interface, empowering other web services to connect with and tap into the shared context service in standard ways.

By way of comparison, Microsoft.NET achieves shared context via the Passport.NET service. Passport.NET is a repository hosted by Microsoft that contains user identity information. It is the cornerstone to Microsoft's Hailstorm services, which is their vision for creating user-centric web services experiences.

In conclusion, we have found that both Sun J2EE and Microsoft.NET support shared context, and each have their own advantages and drawbacks. The noticeable difference is that the Sun J2EE approach of a shared context standard will spawn a marketplace of shared context repositories on the Internet, whereas the Microsoft.NET solution is a single shared context repository approach.

The advantages of the Sun J2EE approach are:

- Each shared context repository can be specialized for different needs. For example, there could be medical repositories that store medical history information, or financial repositories that store credit card and banking information. It is unlikely that a single repository approach such as Passport.NET would be specialized enough to cover all the bases of shared context information that the industry demands when smart web services become widely used.
- There is no 'big brother' effect. Businesses and individuals do not need to trust their data to any individual firm. Local shared context repositories can be created within a trusted few partners in a business web, which means data can be contained.
- We believe the J2EE solution will will scale to handle the needs of the masses. The needs of the many better then Passport.NET. Shared context is a more important phenomenon than any one organization.
- There is no single point of failure.
- There is no control being enforced by a single organization. Can you envision a time when AOL agrees to use Passport? Or when VISA, MasterCard, or American Express agree to let Microsoft control their customers' information? It is difficult to imagine just one vendor-controlled identity service succeeding for all.

The advantages of the Microsoft.NET approach are:

- There is no question of what is the 'official' shared context repository. There is one place to find identity information. This is a very important point. J2EE runs the risk of a fragmented shared context repositories, eliminating the usefulness of such systems, unless care is used.
- Passport is an established and active system.
- Until Sun J2EE standardizes on a schema and API for accessing web services that are shared context services, we do not predict any real usage of J2EE-based shared context services.

Given that almost no web services use shared context yet, it is too early to tell which approach is the best. Fortunately most organizations are barely getting up to speed with the basics of web services, and shared context is in the distance. For most organizations, this debate is likely to be important when choosing J2EE or .NET.

System Cost

A wide variety of implementations based on J2EE architecture are available for purchase, with price points varying dramatically, enabling a corporation to choose the platform that meets its budget and

desired service level. Costs are typically in the single-digit thousands of dollars per processor, although there are higher-end implementations and lower-end ones. At the time of this writing, Microsoft had not released pricing information for the .NET platform.

As far as hardware, J2EE supports UNIX and Mainframe systems, while both J2EE and .NET support the Win32 platform, which is generally the less expensive alternative. There is a level playing field for hardware costs, and the hardware cost debate becomes a moot point.

The takeaway point is that you can get low-cost solutions with both Microsoft and J2EE architecture. Microsoft's solution has an aggressive price, whereas J2EE architecture allows you choose your service level. For example, with J2EE you can have a high-end, expensive solution (iPlanet running on Sun Solaris in an E-10000 server), or a low-end, inexpensive solution (jBoss running on Linux on a Cobalt RAQ server). There's also an assortment of free and/or open source tools and services that support Java and XML[11]. It should be noted that you pay for what you get, and most organizations will not go for this low-end solution, but rather will embrace a midrange solution as a happy medium.

If you are trying to make heads or tails out of the price wars, we recommend that you consider this: the price of the platform is always a drop in the bucket compared to the total cost of the project. This is defined as the price of the server platform, the cost to train developers, the cost to build and evolve a solution on that platform, the cost to maintain the solution, and any business opportunity costs from picking the 'wrong' platform.

We hope that firms realize that the total cost of ownership of a project dwarfs any short-term cost differences between underlying platforms. We recommend you do not consider the price of the platform when selecting between J2EE, .NET, or any other platform, but rather consider the more important other factors.

Performance

A platform performs if it yields an acceptable response time under a specified user load. The definition of what is 'acceptable' changes for each business problem. To achieve acceptable performance, it is important that the underlying web services infrastructure empowers you to build high-performing systems.

The primary bottleneck when building web services is usually integration with back-end database systems. The reason for this is that most enterprise applications are data-driven systems with much more data logic than business logic. Given that the database is usually the bottleneck, any possible tactics for reducing database load will result in a significant wins.

J2EE reduces database traffic through two tactics:

> **Stateful business processes** allow you to maintain business process state in memory, rather than writing that state out to the database on each request.
>
> **Long-term caching** (provided by some implementations) allow for database data to be cached for long periods of time, rather than re-reading database data upon each request.

[11] For examples, check out www.apache.org, www.netbeans,org, www.jboss.org, www.enhydra.org, www.zvon.org, www.juddi.org, www.develop.com/soap, www.alphaworks.ibm.com, or www.themindelectric.com.

It should be noted that both maintaining business state in-memory and caching must be used with caution, and may result in problems if developers are not properly trained on when to (and when not to) use these features. This is a fundamental difference between the J2EE and .NET approaches to building web services: J2EE's advantage is that it gives programmers more control over lower-level services such as state management and caching. Well-educated developers can tap into these features to improve the quality of their deployment. But it is of vital importance that developers are properly educated on when to make these tradeoff decisions, or error may be introduced into systems.

By way of comparison, Microsoft.NET does not offer these tactics for improving performance. There are no opportunities for performance wins, but at the same time, there are no opportunities for developers to introduce errors into systems.

When trying to choose between whether these features are important for your organization, consider the quality of your developers. If they are well-educated and do not require much hand-holding, then they will likely find the flexibility and performance gains from a J2EE system as valuable. If your developers require more hand-holding, then the Microsoft approach is clearly superior.

Scalability

Scalability is essential when growing a web services deployment over time, because one can never predict how new business goals might impact user traffic.

A platform is scalable if an increase in hardware resources results in a corresponding linear increase in supported user load while maintaining the same response time. By this definition, the underlying hardware (Win32, UNIX, or Mainframe) is irrelevant when it comes to scalability, because both J2EE and .NET allow one to add additional machines to increase user load while maintaining the same response time. The major implementations based on J2EE architecture, as well as .NET, provide load-balancing technology that enable a cluster of machines to collaborate and service user load that scales over time.

The significant difference between J2EE and .NET scalability is that since .NET supports Win32 only, a greater number of machines are needed than a comparable J2EE deployment due to processor limitations. This multitude of machines may be difficult for organizations to maintain.

VI. Conclusions

The J2EE verses .NET battle will be the soap opera of the decade for geeks to watch. But there are promises and realities about both platforms. For example, J2EE is a rather brilliant move on the vendors' part, but should not be seen as an altruistic initiative. All vendors that participate in J2EE are after financial gains, as well as an effective weapon against Microsoft. J2EE enables these vendors to collaborate together and stand ground. Many of these vendors have undergone recent mergers and acquisitions themselves, and so organizations must exercise good judgment when choosing such a platform.

As far as Microsoft.NET, that is far from an altruistic initiative. It is a monopolistic initiative dressed in altruism. Microsoft has been claiming that .NET is about open and interoperable web services, when in reality Microsoft is already making their web services closed and proprietary. Microsoft will likely

increase the costs of their solutions if a monopoly can be achieved, and innovation will be slowed down significantly.

So what's a company like yours to do? Both platforms are useful, and both can lead you to the same destination. Which platform is right for you? When deciding, we recommend you concentrate on the larger business issues. Think about your existing developer skillsets, your existing systems, your existing vendor relationships, and your customers. Those almost always drive the decision, not the minor features.

Arguments supporting both platforms

- Regardless of which platform you pick, new developers will need to be trained (Java training for J2EE, OO training for .NET)
- You can build web services today using both platforms
- Both platforms offer a low system cost, such as jBoss/Linux/Cobalt for J2EE, or Windows/Win32 hardware for .NET.
- Both platforms offer a single-vendor solution.
- The scalability of both solutions are theoretically unlimited.

Arguments for .NET and against J2EE

- .NET has Microsoft's A-team marketing it
- .NET released their web services story before J2EE did, and thus has some mind-share
- .NET has a better story for shared context today than J2EE
- .NET has an awesome tool story with Visual Studio.NET
- .NET has a simpler programming model, enabling rank-and-file developers to be productive without shooting themselves in the foot
- .NET gives you language neutrality when developing new eBusiness applications, whereas J2EE makes you treat other languages as separate applications
- .NET benefits from being strongly interweaved with the underlying operating system

Arguments for J2EE and against .NET

- J2EE is being marketed by an entire industry
- J2EE is a proven platform, with a few new web services APIs. .NET is a rewrite and introduces risk as with any first-generation technology
- Only J2EE lets you deploy web services today
- Existing J2EE code will translate into a J2EE web services system without major rewrites. Not true for Windows DNA code ported to .NET.
- .NET web services are not interoperable with current industry standards. Their BizTalk framework has proprietary SOAP extensions and does not support ebXML.
- J2EE is a more advanced programming model, appropriate for well-trained developers who want to build more advanced object models and take advantage of performance features
- J2EE lets you take advantage of existing hardware you may have
- J2EE gives you platform neutrality, including Windows. You also get good (but not free) portability. This isolates you from heterogeneous deployment environments.
- J2EE has a better legacy integration story through the Java Connector Architecture (JCA)
- J2EE lets you use any operating system you prefer, such as Windows, UNIX, or mainframe. Developers can use the environment they are most productive in.
- J2EE lets you use Java, which *is* better than C# due to market-share and maturity. According to Gartner, there are 2.5 million Java developers. IDC predicts this will grow to 4 million by 2003. 78% universities teach Java, and 50% of universities require Java.

- We would not want to use any language other than C# or Java for development of new mission-critical solutions, such as a hacked object-oriented version of C, VB, or COBOL.
- We are finding most ISVs and consulting companies going with J2EE because they cannot control their customers' target platforms. We believe this application availability will result in J2EE beginning to dominate more and more as time goes on.

In conclusion, while both platforms will have their own market-share, we feel most customers will reap greater wins with J2EE. We feel the advantages outweigh those offered by Microsoft.NET. That is our preferred architecture, and we stand behind it.