J2EE or .NET: *A Managerial Perspective*

Neil Chaudhuri
Research Fellow
Logistics Management Institute
McLean, VA 22102 USA

Keywords: *software evolution, J2EE, .NET, project management, enterprise architectures, enterprise systems, web services*

Abstract

With the evolution of enterprise systems from the traditional client-server paradigm, Sun Microsystems' Java 2 Enterprise Edition (J2EE) and Microsoft .NET have emerged as fierce competitors for recognition as the leading choice for building enterprise solutions. After first engaging in a high-level discussion of the architectures, this paper describes the criteria by which project managers should choose between them. It concludes with a discussion of what continued evolution of the enterprise realm, namely towards the web services realm, may mean for project managers.

Introduction

The only certainty in the information technology industry is the rapid and constant evolution of software technology. Nowhere is this more evident than in the realm of enterprise systems, which evolved from client-server systems as a means of physically and logically decoupling the critical components of the architecture—namely the presentation, middleware, and database tiers. Not surprisingly, the size and complexity of enterprise systems demand sophisticated solutions. Assembled from its previously established technologies, Sun Microsystems' Java 2 Enterprise Edition (J2EE) architecture enjoyed prominence as the leading choice for building enterprise solutions. Never to be outdone in its effort to maintain its perch atop the information technology industry, Microsoft Corporation unveiled an alternative enterprise-level solution, .NET, which also represents the next generation of its own previously established technologies. As these solutions have evolved to meet the needs of the evolving enterprise, software engineers on both sides have inundated the literature with perspectives on which is technically superior. To this point, however, project managers charged with overseeing the development of enterprise systems have been largely left out of the discussion and consequently have been unable to engage in any meaningful process of natural selection. This paper provides a comparative overview of J2EE and .NET and describes the criteria by which project managers should choose between them as the solution for an enterprise-level development effort. Finally, this paper suggests how the technologies themselves and project managers' understanding thereof may evolve over time as the enterprise realm continues to evolve--primarily towards web services.

Overview of the Architectures

J2EE

It is the most basic quality of Sun's J2EE architecture that is the initial source of confusion for most project managers. J2EE is *not* a product; rather, it is a *specification*. With each successive specification since the first was issued in 1999, various vendors allied with Sun have built application servers that conform to it, and J2EE applications are deployed to these servers [5]. Among the more notable are IBM's WebSphere, Oracle's OC4J, and BEA's WebLogic Server, which is generally considered the industry leader. During its lifetime J2EE has gained a measure of credibility as a viable option in mission-critical systems, for United Airlines and American Express are two prominent examples among many organizations that have successfully implemented J2EE solutions [7].

With regards to the technical details of J2EE, at its core rests the Java programming language. Over nearly two decades the object-oriented paradigm has become preeminent among programming languages; and in turn Java, another Sun specification, has become preeminent among object-oriented languages. This played no small role in the emergence of J2EE in the enterprise software realm. An even more significant contributor to its marketability is the portability of the Java code that comprises a J2EE application. For example, because both servers meet the J2EE specification, code deployed on WebLogic can be deployed seamlessly on WebSphere should the need arise, and both can operate on any platform (*e.g.* Windows, Linux, *etc.*). Therefore, J2EE does not render an organization vulnerable to the whims of a single vendor. This may be the most powerful argument in favor of J2EE. However, there are some caveats that to be explored a bit later.

While literature concerning the J2EE architecture has been prolific over the last few years, that concerning its components has been even more so. The J2EE specification consists of a potpourri of various technologies, each with its own specification. Moreover, some of these actually predate the first J2EE specification. Therefore, if the age of the J2EE architecture as a whole is an argument for its reliability, the claim is further fortified by the age of its components. The most notable among these are the following:

- **Servlets** and **Java Server Pages** (**JSP**s) for generating web content
- **Java Database Connectivity** (**JDBC**) for storing (or in the vernacular of enterprise architecture, *persisting*) data in databases
- **Enterprise JavaBeans** (**EJB**s) for business logic processing in the middle tier. This is the centerpiece of the J2EE architecture.

As a result of the fragmented nature of J2EE, vendors may produce servers that do not comply with the whole of the J2EE specification but rather with portions thereof. Such servers thus cannot support a complete J2EE application but may still be very useful. A prominent example is Apache's Tomcat, an industry-leading, open-source web server that only supports the servlet and JSP specifications. Hence it is best suited for client-server applications, which remain significant even with the emergence of the *n*-tier paradigm.

Yet despite its excellent performance to that end, Tomcat by itself is quite ill-suited to enterprise architectures.

.NET

While it is designed to solve the same problems as J2EE, Microsoft's .NET architecture takes a starkly contrasting approach to enterprise systems development. The most obvious point of contrast is the status of .NET as a *product* of Microsoft and Microsoft alone as .NET at its core does not rest upon alliances with other vendors. Yet the most significant point of contrast between .NET and J2EE is that the former is virtually brand new. Technically, .NET has been available since 2001, but it has undergone so many modifications since that it is difficult to gauge its readiness as a viable option for mission-critical solutions [3].[1] However, as the giant in the information technology industry, Microsoft has a longstanding reputation, especially within the American government, for producing working solutions. Furthermore, its massive support structure is at the disposal of those who choose to adopt a .NET solution. Therefore, the longevity and stability of Microsoft Corporation goes very far in offsetting the apparent lack thereof in its .NET architecture.

With regards to the technical details of .NET, at the heart of the architecture rests not one but in fact several programming languages. All of the so-called ".NET family of languages" are object-oriented, so an organization need not abandon the paradigm should it choose to pursue this architecture. There are two prominent languages in the family. The first is Visual Basic .NET (VB .NET), which is based on the popular Visual Basic language that has become familiar to countless developers over the last several years. The second is C# (pronounced *C sharp* as in music), a brand new language created by Microsoft. Although the claim is that C# is the next generation in the evolution of the popular C++ programming language, the influence of Java is unmistakable. C# is the centerpiece of the .NET family of languages, and it provides the most effective use of .NET capabilities [1].

The most significant consequence of committing to a .NET architecture is restriction to the Windows platform. Microsoft is often criticized for its reluctance to build products that integrate seamlessly with those from other vendors, and .NET does nothing to assuage the criticism. Yet the Windows operating system represents the very means by which Microsoft ascended to its perch atop the information technology industry. Therefore with the incalculable number of Windows-based systems in operation throughout the world, many organizations would consider a restriction to Windows no restriction at all. Even still, there is a series of open-source initiatives towards moving .NET to other platforms, but they are far from complete [1].

The components of the .NET architecture are the closest point of similarity to the J2EE architecture. The most notable among these are the following:

- **ASP .NET** for web content and based largely on the popular Active Server Page (ASP) technology developed by Microsoft
- **ADO .NET** for data persistence and based largely on the popular ActiveX Data Object (ADO) technology developed by Microsoft
- **Windows Forms** (or **WinForms**) for graphical user interfaces (GUIs) utilized on client machines and based largely on the popular Visual C++ and Visual Basic technologies developed by Microsoft.[2]
- **COM+** (or **Enterprise Services**) for business logic processing in the middle tier and based largely on the Component Object Model (COM) technology developed by Microsoft [4]

From this list one can discern that .NET has hardly emerged from a vacuum. Microsoft is clearly hoping to lure its vast following to its latest innovation by creating next-generation implementations of its prior successes—staples of the industry like ASP, ADO, and COM—and assembling them into its service-based, evolved .NET architecture.

Development and Deployment

J2EE

With the completion of this brief overview of the J2EE and .NET architectures, it is time to explore them in more depth and consider how they compare in development and deployment. The initial step in developing a J2EE application is acquisition of an application server[3], and the best ones, like the aforementioned WebLogic Server, are quite costly. There are cheaper alternatives—including the open-source JBoss available at no cost—but these lack the support mechanisms that can prove invaluable during the course of a project. Ultimately, a leading application server will prove the better value over time despite the heavy cost upfront, but project managers should choose wisely. While the portability of J2EE code across servers is a powerful feature, it is offset by their cost. Indeed, as expensive as a single server may be, to move to another would deplete all but the most lavish budgets. This is simply another case where the reality of the marketplace thwarts the idealism of a technology.

Despite their cost, J2EE application servers provide so many services that they are valuable assets to any development effort. Notable among these is a Java Runtime Environment (JRE), the Sun-specified realm in which all Java applications run. The JRE spares application developers from low-level tasks like memory management which can be excruciatingly difficult to implement. All servers are also endowed with the standard J2EE Application Programming Interfaces (API's) for designing code as well as proprietary API's, which merit particular attention in this discussion.

While all application servers behave according to the standard dictated by the J2EE specification, the underlying implementations are not standard. Thus, the inevitable idiosyncrasies across servers can cause the same code to run faster on one than another. Server vendors therefore provide their own APIs to optimize certain operations like database accesses, and these can lend a rather significant boost to performance. However, these

[1] According to *.NET Magazine*, TRX Travel Services, a provider of reservation-processing services to the travel industry, recently migrated its legacy systems to .NET with excellent results primarily in the areas of scalability and performance [6].

[2] It should be noted that the Java programming language has a similar mechanism in the form of the Abstract Windowing Toolkit (AWT), Swing, and the new Standard Widget Toolkit (SWT). Technically, however, this client-side functionality rests outside the realm of J2EE.

[3] Throughout the course of the discussion on J2EE, the terms *application server* and *server* will be used interchangeably, and both will refer to platforms that meet the J2EE specification.

should be used only when absolutely necessary, for the performance gain comes at the expense of portability. For example, while OC4J database APIs may increase performance by 40%, they will simply not function on WebLogic, and the API-based code would have to undergo an inevitably costly revision if a switch were made. Therefore, heavy reliance on proprietary APIs will all but shackle your organization to a particular vendor's application server, and this negates the single greatest advantage J2EE has over .NET [1]. Project managers must weigh the benefits of both approaches and choose which makes the most sense for the application.

Integral to the services provided by J2EE servers are deployment descriptors, Extensible Markup Language (XML) files which allow critical functionality to be defined without the need for any Java expertise. With only a mere text editor, one can configure essential and otherwise painstakingly difficult services like load balancing and database connection pooling. Moreover, should the requirements for these services change, only the deployment descriptors have to be modified while the code remains untouched. The time that is saved allows developers to focus on the code supporting the business logic of the application rather than that supporting low-level services.

Should a J2EE solution employ EJB's, which is more than likely, deployment descriptors may provide two vital services beyond those previously described. The first concerns data persistence. Charged with this task is a category of EJBs known as *entity beans*. One would think that developers must endow their entity beans with persistence code that utilizes the pervasive but at times complicated Standard Query Language (SQL). Indeed, developers have this option. However, should they so choose, developers may in fact forego writing a single line of persistence code and instead direct the application server to manage persistence.[4] This is achieved by editing proprietary deployment descriptors and specifying data persistence strategies therein. While time must be invested to determine the precise manner in which a particular vendor demands its descriptor to be modified, it is quite easily offset by the time saved by not having to generate the Java and SQL code necessary to manage persistence.[5] Furthermore, the application server will also provide its own optimizations to the persistence strategies outlined in the descriptor. Hence, every effort should be made to have the server manage persistence, for time is saved and performance enhanced as well.

The other significant role that deployment descriptors play in EJB development is in transaction management. Simply put, *transactions* in the context of the enterprise are a chain of operations—almost invariably involving a database—that must all be successful for the transaction as a whole to be considered successful. In that case any database changes made during the course of the transaction are made permanent, or *committed* in the vernacular of enterprise architectures. If even one operation fails, however, the entire transaction fails. In that case all database changes made during the course of the transaction are nullified—or *rolled back* in the vernacular—and the database returns to its original state before the transaction. The concept of transactions is among the most powerful in the enterprise realm, and not surprisingly it is also among the most complex to develop. In a J2EE environment, developers have the option to write code to do this; but as with persistence, they may choose to edit deployment descriptors to call upon the application server to manage transactions. As neither task is trivial, it is quite a boon to developers that they may leave the daunting tasks of persistence and transaction management to the server while concentrating their time and energy on the complex business logic that drives enterprise applications.

When developing an application of any kind, it is necessary to consider carefully which brand of software—known as integrated development environments (IDE's)—will be utilized to write the code that will support it. In the context of J2EE, the situation with IDE's is exactly as with application servers. There are numerous options ranging from free to rather costly, and the number of features available in each is roughly proportional to its cost. Moreover, most organizations would be better served by investing in a leading IDE, for the features it provides will ultimately balance any high costs upfront that may be incurred. For example, while many IDEs like IntelliJ's IDEA provide developer-level functionality like automatic generation of skeleton code, others like Together's ControlCenter also provide architect- and manager-level functionality with Unified Modeling Language (UML) generation and configuration management tools, which may preclude the need for tools devoted to those tasks alone. Yet no matter how sophisticated the IDE, J2EE applications are so complex that development and deployment are hardly ever trivial. A thorough understanding of the subtle points of the architecture is required of all development teams if they are to prevail, and project managers must therefore not presume that investment in a leading IDE will by itself lead to success.

.NET

Development and deployment of .NET are in many ways much simpler matters. Like J2EE, .NET enterprise applications require investment in an application server from Microsoft--most likely Windows Server.[6] Otherwise, there is far less financial investment required than is generally the case for J2EE, for as is custom with Microsoft, the pieces of the architecture are available for free download. Most notable among these are the .NET Extensions to Microsoft's Internet Information Services (IIS) web server and the .NET Framework. The former is a rather trivial matter; as the name suggests, the .NET extensions simply augment the capabilities of the prevalent IIS infrastructure. The latter merits a more rigorous discussion.

Principal within the .NET Framework is the Common Language Runtime (CLR), which is essentially the equivalent of the Java Runtime Environment [3]. The Framework also includes the APIs for developing code in all of the .NET family of languages. The freedom in languages offers great flexibility and tremendous potential for code reuse, but managers should take heed. Having different modules in the same project coded in different languages

[4] It should be noted that although application servers can manage fairly sophisticated persistence code, there are instances when the code is just so complicated that developers have no choice but to write it themselves. Thankfully, these instances are rare.

[5] It is true that switching to another application server would demand the editing of another proprietary descriptor to enable it to manage persistence, but the time loss is probably insignificant when weighed with the benefits, including not having to modify a single line of code.

[6] It should be stressed that Windows Server is only required when utilizing COM+ objects for systems which are truly enterprise-level, the primary focus of this discussion [4]. Smaller systems do not require such an elaborate and somewhat costly infrastructure.

will likely result in a configuration nightmare. Moreover, it will limit accessibility to the code base among the development team. For example, a VB .NET developer will be helpless should the need arise to modify C# code developed by a colleague for the same project. Therefore, project managers should only take advantage of the language freedom provided by .NET in the planning stages of a project and designate a single language as the development standard.

Unlike J2EE, there are few choices for .NET IDE's, and there is a distinct leader in the field: Microsoft's own Visual Studio .NET.[7][8] The tool is expensive, but like the best J2EE development tools, it offers many services like configuration management. Also, borrowing from its successful past, Microsoft endows Studio with both the time-tested drag-and-drop methodology for visually designing applications and GUIs for specifying the properties of objects like the location of a backend database. Moreover, each of these operations results in automatically generated code. Therefore, developers can design a user interface very quickly, and they are spared having to generate the code related to look-and-feel and other more trivial concerns and may instead focus on the business logic. Lest one believe that this is without its cost, however, one must be aware that there are times when it is necessary to understand and modify the generated code to optimize performance, and this may not be an easy task.

As with J2EE, .NET applications contain deployment descriptors, but they do not play a role nearly as significant as that played by their counterparts. Also XML files, .NET descriptors provide the expected services like load balancing and database connection pooling, but otherwise they lack the sophistication of J2EE descriptors. .NET deployment descriptors do not endow COM+ with persistence management capabilities, and this leaves the responsibility for this in the hands of developers [8]. On the other hand, .NET does indeed support declarative (*i.e.* non-programmatic) transaction management, but it is in the form of attributes placed physically in source code files rather than in deployment descriptors or any other kind of configuration files. Although .NET deployment descriptors do not provide the same level of services as those in J2EE, it is reasonable to expect that Microsoft will address this as .NET matures over time.

Choosing Between the Architectures

Understanding the manner in which J2EE and .NET applications are developed and deployed provides the foundation for a discussion of how project managers should choose between them when planning the development phase of a task. There are several critical points to consider, and managers must understand and prioritize them in order to make the right choice.

Establishing the Need for an Enterprise Solution

Foremost among the manager's responsibilities is to determine if the system in question truly represents an enterprise system. This may seem obvious, but it is alarming how often this is overlooked. Part of the problem is that the literature offers no single definition of what constitutes an enterprise system. It would appear that the consensus definition is an architecture comprised of more than two tiers and where each tier may have multiple components (*e.g.* multiple databases residing on different machines). As one might imagine, such a system is terribly complicated and naturally demands the enormous financial and philosophical commitment required by both J2EE and .NET. On the other hand, most systems are not enterprise systems, so it is wasteful to engage in an inevitably rigorous effort utilizing either technology. It is far more sensible instead to utilize individual components of J2EE and .NET—or perhaps even different technologies entirely like ColdFusion. Ultimately, neither a J2EE nor .NET application is trivial to build, so it behooves managers to ensure that the problem is complex enough to merit a complex—and expensive—solution.

Project Funding

Regardless of the project or the chosen solution, it goes without saying that funding is the paramount concern of project managers. Whether pursuing J2EE or .NET, managers can expect to allocate substantial financial resources to training, albeit for different reasons—J2EE because of its numerous component specifications and their rapid changes to meet the demands of the open-source community and .NET because of its own rapid changes in its effort to grow into a robust technology. Aside from training costs, each solution also has similar infrastructural costs associated with it. J2EE demands a large financial investment in licenses for a leading IDE and application server. .NET demands investment in Visual Studio to support application development, Windows Server to support COM+, and perhaps IIS to support web-based interfaces in the unlikely event the organization does not already have it [4].[9] It is difficult to say whose infrastructure is more costly, but it is important to note that these are one-time costs. However, managers who choose .NET at this stage will very likely incur recurring costs in the form of support requests to Microsoft because of the unavoidable flaws in the immature architecture [7]. Given all the variables, only a project manager with knowledge of the existing capabilities of the organization and the development team can decide which is the cheaper alternative.

Existing Client Infrastructure

Project managers must also consider the existing infrastructure of the client when choosing between the architectures. Microsoft solutions in the past have gained wide acceptance in the public sector of the United States. Consequently, public sector clients may not even entertain the possibility of a J2EE solution, and managers will have no choice to make. It would seem logical that the transition to .NET would be trivial, for Microsoft has claimed that legacy objects utilizing older Microsoft technologies will integrate seamlessly into .NET. This is technically true, but there is a significant caveat. Legacy objects imported into .NET run outside the CLR and thus have no access to its services (most notably, as mentioned previously, memory

[7] Throughout the course of the discussion on .NET, the terms Visual Studio .NET, Visual Studio, and Studio will be used interchangeably.

[8] A notable .NET IDE is Web Matrix, a free, open-source development tool that features many of the niceties of Visual Studio. However, it is only useful for the development of ASP .NET applications. Thankfully, web applications that would utilize ASP .NET are so pervasive that the restriction may prove negligible.

[9] .NET enterprise applications may require investment in Microsoft's BizTalk server as well, which serves to integrate the components of the system and is particularly valuable for integrating legacy systems [4]

management) [1]. Unmanaged legacy code, if poorly written, will cause the application to stumble or even fail. Ultimately, legacy objects will almost certainly have to be rewritten as .NET objects [1]. Hence if a rewrite is required anyway, and if the client is amenable to it, it may be advisable to consider a J2EE solution, which as mentioned previously will run on all platforms and thus make the existing infrastructure of the client a non-issue. Whatever the outcome, it is simply crucial that project managers understand that moving from legacy Microsoft solutions to .NET is not as simple a matter as it may seem.

Project Timetable

The factors discussed to this point offer no clear choice between J2EE and .NET because they are a function of circumstances unique to particular projects. However, there are two critical factors where the better choice is much more obvious. The first is the *timetable for completion* of the project. If the project schedule is short, then .NET is almost certainly the better choice. As discussed previously, Visual Studio offers numerous advantages towards Rapid Application Development (RAD). Even the most sophisticated J2EE IDE's cannot compete with Studio in mitigating the complexity of its component technologies and deployment procedures. Moreover, J2EE has a significant flaw in the context of RAD—the elaborate deployment procedures associated with a large, intricate solution are essentially the same as those associated with a small, simpler solution. This makes it difficult to produce systems quickly in J2EE, and .NET therefore has an apparent advantage when time is a significant concern. Of course, it should be noted that this advantage might be mitigated by the expertise of the development team. If an organization only has expert J2EE developers at its disposal, they will almost certainly be able to deploy a J2EE application quickly, and time lost training them in .NET will accrue no net benefit [1]. Thus while .NET lends itself much better to RAD than J2EE, the expertise of the development team can nullify this advantage.

Project Complexity

The other factor where the choice between J2EE and .NET is more obvious is the *complexity* of the project. If the requirements for an application demand a sophisticated solution (*e.g.* multiple servers, multiple backends), then J2EE is the better option. One reason is the ease with which J2EE integrates with multiple platforms. Another is the robustness of EJB's, which by means of deployment descriptors offer persistence and transaction management without the need for a single line of code. Of course, to use all of the features J2EE offers requires significant knowledge on the part of the development team, but when properly implemented these features provide tremendous value to the process. On the other hand, .NET has not yet proven that it has grown enough to meet the needs of a truly complex system [7]. Microsoft has always been somewhat reluctant to enable seamless integration of its products with those of other vendors, and an intricate enterprise system with many components will almost certainly require integration of products from multiple vendors. Moreover, as described previously, .NET's COM+ technology has yet to develop a framework for persistence and transaction management outside the code realm [8]. There is also the issue of Microsoft's poor reputation in the realm of security, which while exaggerated by the pervasiveness of Microsoft products is a significant concern in an enterprise where data are regularly moving over the network. Finally, .NET has only just begun to prove itself as a reliable solution for large-scale, mission-critical systems, and as a result it is impossible to predict just how it will respond to the demands of a particular enterprise. However, Microsoft's stature in the industry all but guarantees that .NET shall have ample opportunities to prove its mettle over time. Therefore, it is only with time that .NET will establish itself as a proven, robust enterprise solution.

Summary Remarks and the Future of Enterprise Software Evolution

During the course of this discussion, a high-level understanding of the components of the J2EE and .NET architectures and the manner in which they are developed and deployed laid the foundation for an examination of the critical points project managers must consider when choosing between them. The analysis led to the conclusion that neither choice is clearly superior in all cases. Rather, as each has its advantages over the other, the suitability of either architecture is a function of the particular circumstances surrounding the project. Understanding the strengths and weaknesses of J2EE and .NET will enable project managers to engage in a meaningful process of natural selection and produce successful results for their customers.

Of course, the very fabric of evolution is woven with the threads of innovation. Thus while the enterprise paradigm--and its implementation strategies in the form of the J2EE and .NET architectures--represent the latest innovation in software development, it is only natural to wonder where evolution will take the industry in the future. It would seem that there may already be an answer: web services.

The Emergence of Web Services

The concept of web services has dominated the literature for some time, yet the prolific rhetoric has actually served to obscure any legitimate understanding of what web services truly represent. Perhaps the best source for an accurate and complete definition of web services is the World Wide Web Consortium (W3C), the standards body that serves as the steward of XML and web services. The W3C provides the following:

> *Definition: A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by internet protocols.* [9]

The goal of web services is *interoperability* among software systems regardless of their underlying frameworks, implementations, platforms, or other idiosyncrasies. This is achieved by communications in the form of XML-based messages transported over networks by HyperText Transfer Protocol (HTTP), both of which are open standards. One can easily see why web services have generated such fervor, for the prospect of integrating disparate systems seamlessly by way of non-proprietary standards is an exciting one.

As the enterprise evolves rapidly towards web services, project managers charged with producing enterprise solutions must have their understanding evolve in parallel. With J2EE and .NET the leading choices for

developing enterprise systems, this paper will now briefly discuss the extent to which they support web service development. Moreover, the purpose of this paper has been to identify the criteria by which project managers should choose between J2EE and .NET as the solution for an enterprise-level development effort. In the interest of completeness, therefore, this paper will also address any additional criteria that must be considered when choosing between J2EE and .NET as the solution for a *web service* development effort.

J2EE Web Services

It would seem that J2EE lacked foresight with regards to the zeal generated by the web services paradigm. As a result the most recent specfication lacks robust support for web services. Transport protocols provide a glaring example. Application servers do not support HTTP as a native communication protocol, so web service requests transported over HTTP must be bridged to another protcol to activate J2EE web services [11]. Ultimately, outside of API's for processing XML utilizing standard interfaces, J2EE is missing a great deal when it comes to web services, and vendors are forced to provide proprietary extensions to fill in the gaps.

The upcoming J2EE specification due for release in the fall of 2003 seeks to rectify many of these problems. The new specification, for example, provides for exposing EJB's as web services for discovery and utilization by clients [10]. Also included is more robust support for processing XML-based messages sent over HTTP [10]. However, as promising as all of this may be, it is all purely theoretical since the specification has yet to be released. The rate at which vendors produce application servers that meet the specification remains to be seen. Moreover, the manner in which IDE's automate web service development to support the new specification also is unknown. Therefore, J2EE developers must currently rely on web service development that is heavily proprietary, and at best they can only be cautiously optimistic for the future.

.NET Web Services

In stark contrast to J2EE, .NET's support for web service development is quite possibly its best feature. Indeed, .NET has from its beginnings demonstrated tremendous foresight in the web services realm. For example, Windows Server has native support for HTTP communication [11]. Furthermore, reflecting Microsoft's commitment to XML, .NET features a rich library of API's for processing XML-based messages. Finally, as one would expect from the powerful IDE, Visual Studio has a number of features to automate the development both of web services themselves and of clients for existing web services.

.NET has proven itself in industry to be an effective architecture for web service development. The aforementioned TRX Travel, for example, has utilized web services to manipulate travel data and to create a generic, reusable interface to its business logic layer [6]. Another .NET web services success story is the Central Bank of Costa Rica (CBCR), who recently ported its legacy application to .NET [12]. Among the web services built by CBCR are a service for messaging, a service for managing financial setlements, and even a service for integrating Java-based financial applications which exemplifies the very interoperability that motivated the genesis of the web service paradigm [12]. The ability to produce such a wide array of web services so quickly has helped .NET to take the lead at this stage in the evolution of web services.

Choosing Between the Architectures

Web services clearly represent a remarkable branch in the evolution of enterprise systems. Though the enthusiasm has been tempered somewhat by the realities of the marketplace, the web services paradigm shall remain a vibrant one[10]. Therefore, as with all enterprise systems, project managers must be able to engage in a legitimate process of natural selection between J2EE and .NET when deciding which will be the architecture for a web service solution.

All of the criteria and considerations discussed previously for a conventional enterprise system still apply when it comes to web services. For example, J2EE's platform independence could be the deciding factor if the platform to which a web service will be deployed is unknown or likely to change. However, there is one signifcant exception to the conclusions drawn previously. Earlier it was suggested that J2EE can quite reasonably claim to be the more proven and more robust solution for an enterprise application. Yet when it comes to web services, the opposite is true. As mentioned before, .NET surpassed J2EE by a wide margin in its appraisal of the web services landscape, and as a result .NET provided web service developers with a great deal of support. J2EE has worked quickly to narrow the gap, but the extent to which it will succeed will only become apparent over time. Thus, with .NET being so far ahead in the web services realm and generally, as discussed previously, being the better choice when time is a factor for any enterprise development effort, it would seem that for now .NET is the better choice when developing web services.

Concluding Remarks

With the enterprise paradigm having evolved from the client-server paradigm and in turn evolving in some measure towards the web services paradigm, project managers must contend with many complex issues when choosing between J2EE and .NET. What makes their task even more difficult is the rapid pace with which the architectures themselves are evolving in an effort to become more robust. This paper has attempted to articulate and clarify the criteria that project managers must consider when making their choice.

It is unclear if natural selection by the marketplace will ultimately determine a victor in the battle for the enterprise between J2EE and .NET. Rather, it is far more likely that the two will simply coexist in the ecosystem of enterprise architectures. In fact, we can make only two claims with any certainty: the rivalry will continue to make for fascinating theater, and more importantly, the true victors will be project managers and their development teams, all of whom will reap the benefits of the evolved functionality that will inevitably result from the competition.

---

[10] As true interoperability among systems is an extremely difficult goal to achieve, web services have come to encounter resistance in several forms, including the rapid evolution of standards (especially in the area of security of XML-based messages), the deliberate pace with which vendors adopt new standards, database concurrency issues [13], and the performance cost of XML-based messaging. The W3C must address these and other problems—and vendors must adhere to its recommendations—if web services are to continue to flourish.

Acknowledgements

The author would like to recognize the following individuals who helped to improve this discussion: Geoffrey Simpson, Mauricio Calabrese, Randa Khoury of the National Academy of Sciences, and Jonathon Leete and Sam Stange of Logistics Management Institute (LMI). The author would also like to express his heartfelt gratitude to Vice-President Dr. Susan Marquis and Program Director Robert Hutchinson of LMI, whose continuous encouragement and support were invaluable. Finally, the author wishes to express a special note of thanks to John Kupiec of LMI for his mentoring guidance, unwavering patience, and sage wisdom. This paper would not have been possible without him.

References

[1] McAllister, Neil. *New Architect*: "The Great Migration: The Rocky Road to J2EE and .NET." March 2003.

[2] Roman, Ed. TheServerSide.com: "A few tips on deciding between EJB and COM." Unknown date of publication.

[3] Lowy, Juval. *.NET Magazine*. "Set a New Course With .NET" December 2001.

[4] Sessions, Roger. Java 2 Enterprise Edition (J2EE) versus The .NET Platform: Two Visions for eBusiness. March 2001.

[5] Marinescu, Floyd. TheServerSide.com (www.theserverside.com): "The State of The J2EE Application Server Market: History, important trends and predictions." March 2001.

[6] Bustamante, Michèle Leroux. *.NET Magazine*: "TRX Travel Services Goes Live With .NET." May 2003.

[7] Hatem El-Sebaaly. UC Irvine Extension (unex.uci.edu): "J2EE vs.Microsoft.NET: Choosing an Enterprise System." August 2002.

[8] MacHale, Robert. Microsoft Developers Network (msdn.microsoft.com): "Distributed Transactions in Visual Basic .NET." February 2002.

[9] Champion, Michael; Ferris, Chris (eds.) *et al*.: *Web Services Architecture*, W3C Working Draft, November 2002.

[10] Varhol, Peter. *JavaPro Magazine*. "J2EE 1.4: A Web Services Kit." August 2003.

[11] Newcomer, Eric. *.NET Magazine*. "Decide Between J2EE and .NET Web Services." October 2002.

[12] Thé, Lee. *.NET Magazine*. "CBCR Ports Critical App to .NET." August 2003.

[13] Ambler, Scott. Lecture: "Agile Database Techniques – Data Doesn't Have to be a Four-Letter Word Anymore." August 2003.