

The Deliberate Revolution: Transforming Integration With XML Web Services

From [Building Web Services](#)

Vol. 1, No. 1 - March 2003

by **Mike Burner, Microsoft**

The vast investment in Internet infrastructure and telecommunications over the past decade is making the unthinkable eminently achievable. Organizations can now retrieve up-to-the-minute data at run-time from its canonical source, partners and customers. And where applications have traditionally bound functionality together, it now is practical to access application logic at run-time, from hosted services updated dynamically to keep current with evolving business processes.

Parties must now agree on how to represent information, on protocols for retrieving and updating data, and on means of demonstrating the privilege to do so. Such are the necessities that gave birth to XML web services. The architecture of these services attempts to bridge myriad Internet systems, to let organizations ranging from small businesses to multinational enterprises to world governments communicate more effectively using programmatic interfaces to data and processes.

Web services have generated much excitement, and vendors are scrambling to depict their platforms as the most compliant, mature, secure, or simply the most likely to crank out swell T-shirts. This article attempts to dive beneath the hype, examining how XML web services differ from existing architectures, and how they might help build customer solutions. Let's begin by describing features of XML web services, which:

- *Expose programmable application logic.* A client application calls a web service, often providing input parameters, to retrieve a set of results it will take further action on. Scenarios run from retrieving the phone number of a local restaurant to participating in the search for extraterrestrial intelligence. Like a call to a class library or analytic engine, you use web services when you cannot, or would rather not, implement the logic yourself.
- *Are accessed using standard Internet protocols.* At minimum, this means utilizing TCP/IP or UDP, but web services are usually exposed using HTTP operating on top of TCP. More on the use of HTTP in the next section.
- *Communicate by passing messages.* A web service is defined by messages it accepts and produces. Conceptually, these messages must be self-sufficient, containing or referencing information necessary to understand the message. In practice, part of this message state may be implied: when using HTTP, the service reply is interpreted as responsive to the request sent on the same connection.
- *Use XML to structure messages.* XML (eXtensible Markup Language) is a mature technology for representing data as self-describing, platform-independent text. Self-describing means several things: first, that data in an XML document identifies itself using element and attribute names, and second that elements identify their type, such as "integer," using the XML Schema Definition Language (XSD). XSD allows services and clients running on diverse platforms

to interoperate over a common type set, and is critical to the success of web services.

- *Package messages according the SOAP specification.* "SOAP" is an acronym for "Simple Object Access Protocol," but that expansion implies a programming style inconsistent with the document-centric style growing dominant in the web services space. SOAP is a simple protocol that, among other things, defines a message structure to include an optional Header element and a mandatory Body element, wrapped by an Envelope element. While simple in itself, SOAP supports the creation of complex self-contained messages; the pros and cons of SOAP represents are explored more deeply later.
- *Describe themselves using WSDL.* The Web Services Description Language (WSDL) allows a service to define the messages it accepts and produces; collectively, these messages define the service contract. WSDL also permits the service to identify network endpoints that honor this contract. The WSDL document containing the service definition is typically retrieved from a URL using HTTP, but may be transmitted by many protocols and means.
- *Support their own discovery.* If you can find a WSDL document, you can use the web service it describes. But how do you find the web services useful to you? How do you find the service on your local area network, rather than the one at headquarters halfway around the world? UDDI (Universal Description, Discovery and Integration) supports both design-time (used by the client solution developer) and run-time, or dynamic discovery of web services. UDDI is implemented as a web service, and you run it within a trust domain to access the services important to your organization or consortium. Several public implementations of UDDI are operated by member organizations of the UDDI community to provide an Internet-scale directory of businesses and services.
- *Are not remote procedure calls.* Now to stir the pot a little: good web services are not modeled as remote procedure calls (RPCs). Section 7 of the SOAP 1.1 specification describes how to express a remote procedure call in the Body element of a SOAP message. This is a useful mechanism for tunneling RPCs between systems, but SOAP-RPC is a poor approach to designing an interoperable web service. RPC is primarily designed to support object invocation between tightly bound but topologically distributed systems. Yes, you can still achieve interoperability while using "RPC/Encoded" SOAP messages, but the focus on objects and method invocation is fundamentally contrary to the document-centric philosophy of web services. SOAP messages can be in two styles, "RPC" and "Document," and can use two serialization formats, "Encoded" and "Literal." In practice, SOAP-RPC uses "RPC/Encoded," while document-centric web services use "Document/Literal." By failing to focus on the service contract, you provide for brittle integration. Minor changes to a method signature automatically get propagated to the service, causing current clients to break. Great toolkits exist for doing tightly-bound RPC over SOAP from various vendors. If this is what you are looking for, find the one that works best for your platform. But if you are trying to design for interoperability, design your messages first and then write your methods to support them, not the other way around.

- *Is not CORBA.* Making the messages and the service contracts the design center of web services is the fundamental difference between the web services architecture and CORBA. There are strong analogies between elements of the two architectures, such as the respective roles of WSDL and IDL, but CORBA is fundamentally object-oriented. The messages CORBA passes are manipulated by instantiating an object. The document-style messages used in web services offer more flexibility for manipulation; for example, an interception service (more on this pattern later) might operate on one document header element, without having the logic to understand the rest of the document. As the technologies of web service rapidly evolve in the years ahead, standards bodies, including the W3C and IETF, will be instrumental in smoothing the rougher edges of the web service specifications. But these technologies are mature enough—and widely adopted enough—to be actionable today.

[MIKE BURNER](#) is a software architect at Microsoft, working on Web services. Mike's recent work includes collaboration technologies, Web-based storage services, and participation in the specification of .NET My Services. Prior to Microsoft, Mike worked at Alexa Internet developing Web-profiling technologies, Web-service based syndication technologies, and the Internet Archive. Mike's fascination with integration and interoperability comes from his early career as a Unix systems programmer and system manager at Harvard University and Xerox, where his days were spent getting Unix flavors, VMS, CMS, Macs, PCs, and Xerox Network Services to play nicely with each other.

A Conversation with Adam Bosworth  
From [Building Web Services](#)  
Vol. 1, No. 1 - March 2003

**Adam Bosworth's contributions to the development and evolution of Web Services began before the phrase "Web Services" had even been coined. That's because while working as a senior manager at Microsoft in the late '90s, he became one of the people most central to the effort to define an industry XML specification. While at Microsoft, he also served as General Manager of the company's WebData organization (with responsibility for defining Microsoft's long-term XML strategy) in addition to heading up the effort to develop the HTML engine used in Internet Explorer 4 & 5. Now, as Chief Architect and Senior Vice President of Advanced Development at BEA Systems, Bosworth is much more directly involved in shaping the future of Web Services.**

**To press Bosworth for insights into the possibilities and hazards he sees ahead, Queue asked Marshall Kirk McKusick - former head of the UC Berkeley Computer Systems Research Group (CSRG)—to fire off a few questions regarding his greatest Web Services concerns. Besides overseeing the development and release of 4.3BSD and 4.4BSD, McKusick is also renowned for his work on virtual memory systems and fast file system performance—and in recent years he's also achieved prominence as one of the leaders of the Open Source movement.**

KIRK MCKUSICK (KM) People sure talk a lot about Web Services, but it's not clear they're all talking about the same thing. How would you define "Web Services"?

ADAM BOSWORTH (AB) The term Web Services refers to an architecture that allows applications to talk to each other. Period. End of statement.

KM Fair enough. So what can we say Web Services aren't?

AB Well, they aren't super-efficient. But that may not be such a big deal since we're talking about self-describing messages that are easy to route and control and massage along the way. And that's something that wasn't true under previous message infrastructures.

It's also true that Web Services are not appropriate in every case for B2B today because reliable and secure delivery really requires some other transport. And clearly, Web Services don't offer a high-performance broadcast model just yet, which means they're not the best way to provide for real-time data delivery. The fact is that, in designing Web Services, we've been more concerned with an even more basic question.

KM And what would that be?

AB On the Net, we have a lot of applications that need to talk to each other, which means we've essentially moved from a two-tier world to an n-tier world. So the key question is: What is the right architecture for n-tier communication? We've come up with three basic principles that we've stuck to like glue. And those have driven our approach to Web Service design from the start. They're also just as relevant today, which is why I think Web Services have really begun to take off.

KM And what are those three principles?

AB They're not all intuitive. The first, of course, has to do with communication efficiency. Let's say that, via the Web, you wanted to pull information out of a set of SQL health records containing hundreds of different entries. Obviously, you wouldn't want to be forced to query initially for the first field of the first row and then for the second field of the first row and so on and so forth through every single item contained in the records. That wouldn't even work for a local database application, never mind for the Web. Because, with an approach like that, it wouldn't be long before you shut the database down. In the client/server world, the database has a fully maintained stateful TCP/IP connection with its client. But we don't have that on the Web. Instead, we have stateless protocols. So it's more expensive for each one of these round trips. Also, whereas in the case of the local database we don't necessarily have a cross-platform problem because there's code on both sides to ensure that the exchanged binary chunks line up properly, we really do have to worry about cross-platform issues when you're looking at application-to-application communication.

There also are transactional integrity issues to consider. A database can maintain a transaction over a short period of time in a client/server setting. But if you have one application going to another to ask for 1,000 different things at 1,000 different points in time, the odds that consistency will be maintained are pretty slim. So what would make a lot more sense would be to ask for everything you want from that medical record at the same time. That way, it's just a single transaction over the Web. And then inside the hospital system, they can pull together all the needed data and hand it back as one big screen chunk. Now that model is very efficient for the Web and it doesn't require you to maintain state. Basically, you've got to take a coarse-grain approach like that or your system just isn't going to scale. That's true in the database world and it's even truer in the application world.

KM So far you've talked about communication and server efficiency. What are the other two principles you referred to?

AB The second has to do with loose coupling. Think of all the applications out there as a sea of bubbles. That way, you can see any attempt to integrate applications as a matter of drawing together some of those bubbles and possibly even acting as a source for some others. Over time, some of the bubbles are going to change since people are constantly evolving applications. They redeploy, they extend, they modify. They may even replace classes or swap operating systems. Not all of the bubbles are changing, but over time, you're going to see a lot of movement. And if your architecture isn't resilient and doesn't help applications to keep working with each other in the face of all that change, it will break. And that means every single change to every single application could become a point of instability for the entire system. So unless your architecture is capable of reliably handling changes gracefully, you can be sure it won't be suitable for the integration of applications.

For example, we failed that test with client/server computing. Because when you have code on your client that talks to your database, the odds are high that when you need to make a change to that database, you're going to end up having to redeploy your client code — as anyone who's ever built client/server applications has discovered shortly after modifying the server.

Now, in the world we used to live in, client code generally lived on employees' desks, so when it broke it was bad — but it wasn't really all that terrible. You just took your new code, redeployed it on all your employees' desks and life went on. But if we're talking about applications that flow across the Internet, there's no way you can go and redeploy every other application just because you change your mind about how your app is supposed to work. So applications can't be designed in such a way that they depend on the implementation of other applications. There has to be absolute confidence that you can change the implementation of your app without breaking others. The funny thing is that when we did object-oriented computing, we assumed we were solving this problem by adding properties and encapsulations. So you didn't actually know how data were being stored because they were encapsulated and hidden through methods. And we assumed that protected us from implementation changes. We also assumed we could deal

with change because we had interfaces that we thought would support both a new interface and an old interface whenever you evolved an object. So, rather smugly, we thought: Yes, we've cracked the code. We now have an evolvable object model.

KM But we found out otherwise, didn't we?

AB Yes, it turns out that there actually are two problems. An object is stateful and you have very fine-grained access precisely because you also have encapsulation. And what we learned through hard experience is that you can't just call any method or set any property any time you want to. There are all sorts of rules that dictate how you talk to an object—providing choreography of sorts that you really have to observe. If you should ever change the order, you'll often find that things don't work at all. And no one really knows what the rules are because of the complexity of the interfaces, which are so fine-grained in the state machine that they'd be much too hard to describe and far too hard for anyone to understand. But given that you don't know what the rules are, it's as though you rented a house with a big fat lease full of fine print you couldn't possibly read or understand.

So that's bad news. We've had two major revolutions in computer science over the past 20 years: client/server computing and object-oriented computing. And they've both flunked this test—and it's a test we have to pass if we're to have architecture suitable for application integration.

KM So what do you do?

AB We looked at the Web. Of course, that was a natural for me, since I ran the team at Microsoft responsible for building the HTML engine for Internet Explorer 4 and 5. It also turns out that a Web browser is an extraordinarily successful example of a program that talks to other programs called Web sites—which, of course, are prone to change all the time. In fact, the databases, the classes, the operating systems and just about everything else involved in a site are almost guaranteed to change over time and yet we still expect the browser to keep on working perfectly with the site. And the reason that's true is the browser knows absolutely nothing about a site's implementation. It knows only two things: wire-level protocol, HTTP, and a set of wire-level formats—most particularly, HTML. And as long as the site supports that protocol and some known format, the browser is going to continue working.

The other reason browsers took off was their ease-of-use. HTTP is a terrible networking protocol. It's inefficient. It's clumsy. But it's also quite easy. Anyone can implement it, which makes the Web a democracy. Nobody's in control, and so what happens is that some of the things that are implemented finally hit a tipping point where everyone just starts using it because, hey, it works, everybody else seems to be using it and you get a network effect as a consequence.

So we looked at browser architecture and figured we should use essentially the same model to create a wire-level protocol for Web Services — and that ultimately, of course,

came to be known as SOAP. And then there also were the wire-level formats to consider, which are described by this thing called WSDL, whose job it is to enumerate XML messages.

KM Why XML?

AB We ended up going with XML because we felt it gave us a good way to describe coarse-grained messages—important for all the reasons I mentioned earlier. And we asked ourselves what would be easy, in the public domain and flexible enough to handle anything from a highly structured coarse-grained message (such as a purchase order) to something far less structured (like a doctor's write-up incorporating certain data from a patient record).

XML gives you a coarse-grain solution that allows for communication efficiency. SOAP and WSDL, meanwhile, give you the loose coupling you need. But that's only true if when you change your implementation, you make sure none of your XML messages are changed, because that's where your public contract is established. Imagine, for example, that I changed my Web site such that it no longer used the HTML format. Let's say it did WML instead. My browser wouldn't end up being such a happy camper if I did that, would it? And in the same way, if I were to change an application such that it didn't do a particular grammar of XML anymore, the other applications I have to communicate with would suddenly get very unhappy. So it's critical that the thing in charge here is the WSDL and not the code.

I bring that up simply because a lot of people building so-called Web Service solutions do it just exactly the other way around. They have you build your code and then they auto-generate the XML messages from a description of that code. But, sooner or later, the code is going to change. And when it does, all those auto-generated XML descriptions are going to start breaking things.

KM Well, that certainly seems to cover “loose coupling.” But I believe you also mentioned a third principle?

AB Yes, and that has to do with asynchrony. In the real world, you have to be prepared for those instances when applications will not be available. And there are three basic reasons for that:

Number one, applications go down. Everyone knows that no application is available 100 percent of the time. My brokerage firm, for example, won't tell me about how my stock portfolio is performing between midnight and two in the morning on Sundays. And when you find yourself faced with sort of thing, you can't just sit there blocked, waiting for your application to return. So you have to have a way to handle that.

The second major cause of unavailability is that certain applications may not be capable of handling a given load. That's because most applications weren't written for a world of unpredictable loads. The very reason BEA even exists, in fact, is because the Web

opened up a whole Pandora's box of unpredictable loads. As a consequence, you can pretty much expect to run into temporary bottlenecks from time to time. And as you try to spin out more and more threads to talk at the application level, they're apt to block. Or worse, the application may itself block or thrash or fail. So it's critical to keep in mind that most applications can only sustain a certain number of requests in any given period of time.

The third major cause of unavailability is that people often ask applications to perform tasks that simply can't be completed immediately. There's a reason, for example, that Amazon.com doesn't tell you immediately that the book you've ordered has been shipped out to you. That's because a human being first has to go out and retrieve that book. It may not even be in stock. So, as a consequence, the initial response merely tells you that your order has been received.

KM So then, how do you provide for all those instances when applications will simply be unavailable?

AB Basically, you build a message-based architecture. And that was certainly something we kept in mind as we designed our approach to Web Services. One of our reasons for going with XML messaging and the SOAP protocol was that they not only provided for synchronous invocation but also for the sort of message-packet exchange we needed to enable asynchronous invocation. And that's really one of the keys to the whole BEA approach. We support both synch and asynch because we think an enormous amount of application integration ends up being inherently asynchronous, if only to allow for robustness in the face of unpredictable loads, application failures and various other availability problems.

KM And don't you need to be stateless as well as asynchronous?

AB Right—as stateless as possible. We don't always make things stateless because there are cases where you simply can't – or rather, if you did, you'd only be pushing the database program out to someone else so that they could store state in the database for you. Anyway, those were the major principles we had in mind when we set out to design Web Services. And already, more and more people are beginning to take advantage of that to wire together applications that need to communicate—for the time being, mostly within the enterprise, where the security issues aren't nearly as difficult and there already are available protocols and reliable delivery transports. And that's what we call “queuing.”

We're also starting to see some application integration across enterprises. And over the next six months to a year, I think we'll start to see better security provisions for that, along with some protocols that extend SOAP for greater reliability. In fact, I expect to see security improvements in the very near future.

KM I'm glad you mentioned security. Is there an architecture in place already—or at least a roadmap for getting to where we need to go?

AB There is indeed an architecture for security. In fact, a spec will be coming out soon that describes in great detail exactly how that architecture is supposed to work. The thing to understand, though, is that the spec recognizes that security today works according to a federated model, not a centralized model—and that most people would like to keep it that way. That's to say that the notion of Microsoft as a single source of security for every single transaction in the world is not an idea many people are ready to embrace. And, in general, history has shown that things tend toward decentralization in any event—whether we're talking about biology, physics or software.

KM So tell us a bit more about this federated model of security.

AB The first thing you want to be able to do when a Web Service request comes in is to establish the credentials of the sender. That is, you want to authenticate that the request in fact comes from the person it says it does. And our emerging security architecture certainly is capable of supporting that. You also need the ability to decrypt because the messages we now have in XML provide for encryption and decryption as well as for signing. And we have assertions that will allow you to authorize certain actions on the basis of pre-established permissions levels. So when you authenticate, you should be able to figure out the requestor's role and then use that information to determine whether they're authorized to do whatever it is they're asking to do. Now, as we all know from painful experience, no security system is going to be perfect right out of the gate. It's a hard problem. But there most certainly is an architecture that's being put into place for Web Services, piece by piece, at the XML layer and the WSDL layer.

The other we're seeing is a rapid increase in the demand for message brokering and Web Service management. Essentially, as corporations start opening up all their resources to Web Services, they're asking how they can control who can do what. Furthermore, how can they provide for logging? How can they monitor? The issues of command and control and routing are becoming paramount, so over the next two years we expect to see an enormous groundswell of activity invested in the development of management tools designed to let corporations better control all of these requests floating through the system. And I think we'll see more and more of that until metadata begins to play a central role in just about every invocation of application interoperation.

KM How does BPEL fit into this scheme?

AB Well, it turns out that the advent of message-driven paradigms is driving a requirement for workflow. BPEL basically allows you to script that workflow. And to understand why that's important, let's look at Visual Basic for an analogy. One of the great strengths of Visual Basic is that it gives you something almost anyone can use—a form designer. And something a programmer, a systems programmer, or even a non-programmer can employ to indicate how an application should work.

Likewise, to design and control workflows, you need a visual designer that even mere mortals can use but which also incorporates some solution that systems programmers can use to extend these models (creating what we call “adapters”). But the question is: What

happens when messages come back to say that some additional procedural action is required? How can mere mortals be expected to deal with that? Our customers want an answer there because that would effectively make workflow available to the mass market. But first we have to have a standard. And that's very tricky because ultimately you're describing something that will extend the whole programming model. BPEL is the result of an effort by Microsoft, BEA and others to start solving that problem — which is to say: how to provide a standard model for writing workflow?

In terms of implementing that, the plan here at BEA is to essentially use metadata to drive the required extended programming for workflow semantics so that the programming language for our customers will still be Java. And that's largely because we don't think customers really want yet another programming language—let alone one described in XML grammar.

KM How does that compare to the .NET approach? My sense is that the .NET philosophy might best be summarized as “any language, one platform,” whereas the Java approach is more a matter of “one language, any platform.”

AB Back when I worked for Microsoft, I built complex infrastructures for customers. I was quite proud of that work because I felt we'd succeeded in bringing together all the tools our customers needed. We'd given them Visual Basic to build forms, and we'd given them active server pages to build pages, and we'd given them XSLT to do conversions between XML and HTML. And we'd given them C to write code, and so on and so forth.

But then I had an opportunity to meet with a lot of customers, who explained that it's incredibly hard to train people and—all things being equal—they'd just as soon train them in only one language. And almost without exception, they told me that's just exactly why they found Java so appealing. They said that, in their view, Java had finally gotten to a point where it had enough power to satisfy the average systems programmer. And yet, it also managed to hide most of the complexity that's historically made something like C a very tricky language. Garbage collection, for example, is something that Java just automatically handles for you. The same thing holds true for multiple inheritance. So that effectively gave them one comprehensive solution, and they just loved that.

At the same time, I don't know of many customers that have just one platform. So it would be arrogant for us to say we didn't feel we needed to make our product cross-platform. The value of cross-language, on the other hand, is much less clear. In fact, for most of our customers, it's as much a curse as a blessing. And that's because issues tend to arise when all your programmers are using different languages in different ways. Now if Java were intrinsically a hard language, or an inherently limited one, I think there would still be a good argument for having multiple languages. But Java is intrinsically a pretty easy language. The hard thing about learning Java isn't Java itself. It's J2EE and all the plumbing required to build highly scalable transactional applications. And frankly, we've been investing a lot of our time here trying to make that a lot easier.

So the .NET idea about many languages being a good thing, I believe, is quite open to debate. Now, bear in mind that I came from Microsoft and still have the highest respect for the engineers who built .NET. But I've yet to hear of a customer problem that was solved as a consequence of having multiple languages. And I've heard of plenty of customer problems that have been caused by having multiple languages. So I guess you'd have to consider me a bit of a skeptic.

KM Well, let's say you're right about that. But .NET does come from Microsoft, and Microsoft does exercise a fair amount of market clout. Can't they just essentially ram .NET down people's throats?

AB Microsoft doesn't drive the market when it comes to enterprise computing. What they've really done is create an alternative, which I consider healthy. It's making the J2EE people over at Sun wake up and evolve their capabilities a lot faster. For the customer, this is nothing but good news. In any case, what it really all comes down to is how you handle the Web Services stack. And the truth is both J2EE and .NET still have room to grow on that account.

What might be more germane to your question is that, for all the clout Microsoft wields, they're still trying with mixed success to extend their reach into the enterprise world from their long-established stronghold in the desktop world. J2EE, on the other hand, is already widely used by almost every Fortune 500 company to deliver just about every mission-critical application you can imagine. And we also know that enterprises are using J2EE on their Unix and Linux and mainframe platforms, because they're certainly not using .NET for that. In fact, I think you have to wonder what will become of .NET if Linux should someday become ubiquitous. As you suggest, history has shown that at the end of the day, there tends to be only one winner in the software standards wars. And right now, while NT is obviously a huge factor in the enterprise computing space, it's my sense that Linux is growing much more rapidly. And, if that continues to be the case—with J2EE being a natural partner to Linux—I'd have to think that .NET is perhaps in a world of trouble.

KM You've already noted the power of software standards. So my question is: Who is going to set the standards in this space? Are we going to see standards established du jour by groups like the W3C and IETF? Or do you foresee de facto standards set in the marketplace?

AB At BEA, we've generally taken a two-pronged approach to standards. First, there are those that address how you implement code—the Java Community Process, for example. We worked very closely with JCP to address the issues that we felt were relevant to our coding world. But where we tend to get a bit more creative is when it comes to cross-platform issues. And that's where we recognize that there are two companies that are absolutely central to any notions of interoperability. Those two, of course, are Microsoft and IBM. If you can't interoperate with Microsoft apps and IBM apps, how can you credibly claim to offer interoperability? And conversely, if Microsoft and BEA and IBM should get together and decide that something really makes sense, then you're obviously

looking at critical mass, since most Web Services applications are built with tools that come from one of those three vendors.

Plus we're realists. It just makes sense to work together with IBM and Microsoft to develop standards that are likely to benefit all our customers. All three companies are strong enough to believe they're each going to end up with a healthy slice of the pie. But first, there's got to be a pie, and what we're talking about here is application integration, which inherently requires standards. So in order to turbo-charge the pie, we've got to have standards that are going to be widely adopted.

KM Where is it that you are most keenly interested in influencing those standards?

AB At the highest level, we've been doing a lot of thinking about asynchrony. Specifically, how do you make asynchronous Web Services work well? And we've also been doing a lot of thinking about reliability.

KM But what are you going to be most closely identified with? It's obvious to everyone that Microsoft promotes .NET and Sun pushes Java. But what flag is BEA waving?

AB We are the poster-child for J2EE. We're the original J2EE application server and we're still by far and away the best J2EE application server. IBM is waving the J2EE flag as well. But what sets us apart is that we're focusing on the innovations required to make it easier. There are, roughly speaking, 10 million people today who write code and probably less than a million of them are really productive in J2EE right now. We're changing that by seeing to it that everyone who's a developer can actually work with it.

KM And is J2EE actually robust enough to stand up to the rigors of production environments?

AB Are you kidding? J2EE applications stand up extremely well. When it comes to robustness, bear in mind that most of our customers are running mission-critical applications. J2EE is not the issue. The issue for our customers is how an application evolves. If you build and deploy an application that's running 24x7x365 worldwide and can never really be brought down, you've got a problem when the time comes to extend and modify it. So that's where we're focusing a lot of our attention right now—on making it easier to evolve running apps in a controlled and manageable way, without ever bouncing them.

KM And how do you manage that?

AB We're doing it in two different ways. We've made some huge strides in the area of incremental code development so that if you make a change to a single class, you don't end up having to re-deploy the entire application.

And we're also putting a lot of effort into metadata so as to make more and more of what we do describable and modifiable. In particular, we're working on how to deploy

metadata across the enterprise and we're working on transactional ways to re-deploy metadata across the cluster. The idea is, as that happens, the application's behavior should evolve without the need to introduce new code.

KM When I've looked at trying to accomplish those kinds of things, I've found you generally need a Level 1 programming wizard to manage all the details. How are you handling that?

AB I can't divulge everything we're doing, but let's just think for a moment about the heritage of J2EE. Why does J2EE have containers? By and large, it's because it was considered a bad idea to have programmers write the low-level plumbing involved in determining whether you were stateful or stateless and whether you were transacted or not. Those models were hidden in the containers as metadata. You write deployment descriptors and the deployment descriptors in turn tell the container what to do so the developer doesn't have to write all that code.

In some sense, J2EE started off on this footing. It started with the idea that you could change deployment descriptors in a deployed application, knowing that incremental changes would then just ripple throughout. So the heritage of the application server and the J2EE platform itself is one where metadata describes things and the metadata itself is open to change. The tricky thing is that it's really hard to write the containers. That is, it's really hard to write the code that takes advantage of the metadata, and that's where we're focusing our attention right now by writing a paradigm for Web Services and a paradigm for components that abstract away a lot of the plumbing you'd otherwise have to build yourself. That leaves you to administer and alter the metadata without having to think through the plumbing. And, yes, what we're developing is extremely hard to write. But when you do get it right, you save everyone else all that work.

KM OK, so let's assume all of this works out wonderfully well and Web Services end up proliferating just as predicted. What are some of the messes we can look forward to cleaning up, say, five years from now?

AB One of the messes will almost certainly have to do with what I'll call Web-wide garbage collection. Code creates objects and at some point that code dies. And when that happens, the objects become garbage that needs to be collected. Now, Apache noticed that didn't always work out all that well—and that systems got sick as a consequence. So they started killing processes periodically as a weeding technique. That turned out to be brilliant and it's one of the main reasons Apache proved to be such a good Web-based container for such a long time.

So one of the techniques we're going to have to implement is a way for all the invoked resources that have been orphaned to be gracefully cleaned up. As we move to a message-driven model, we're going to find that Web Services start processes that run longer and longer and are more and more expensive across the Internet. And sometimes the roots of those processes will go away. And there is no clean way today to tell all the children they need to go away as well. Right now, we have no idea of exactly how that's

going to work. And the problem is certain to be compounded by a proliferation of B2B and mobile devices. So it's going to be really challenging to handle all the in-flight data that's going to be created. And, here again, you're looking at processes that may end up trying to talk back to systems that are no longer there. That's going to be tough.

KM How about languages?

AB Definitely, another challenge has to do with language. We don't have a good language today for dealing with XML and that's a real problem. We have more and more systems that use XML extensively—either as metadata that either describes what to do or simply transmits data from one application to another. The first step in any of these exchanges is called “binding,” which involves some tricky processing to turn the XML back into data structures the programming languages themselves can understand. So if you send me a purchase order in XML, the first thing I'll have to do is tell you how to turn it into a purchase order object. Now, we've already invested a lot of work into ways of handling that here at BEA and we think we've done a pretty good job. But ideally, you shouldn't have to do any of that at all. What you'd really like is for the language to be able to understand the XML document and extract the necessary information itself. In fact, ideally, the language would do even more than that. Because these messages are self-describing, you should also be able to query your own data structures. If someone sends you an XML document, you may want to query it to find out what things you want. And we don't support that today because languages aren't used to thinking about their own data structures as query-able objects.

So I think the changes that are going to be driven by Web Services will result in a major language extension. And that will give us a language that not only understands the idea of self-describing documents but also actually is capable of querying them and treating them as data structures.

KM Are you talking about an extension to XML or the emergence of an entirely new language?

AB Whole new languages come around very rarely. So, ideally, from our point of view, we'd like to see this come about as an extension to Java. And we are seeing some of that happening already, which is just as well since our customers really don't want a new language. It goes back to a basic principle about design that Alan Kay once articulated. He said that simple things should be simple, while hard things should be possible.

In the real world of building applications, there are three kinds of people who come together. There are the systems programmers, who are very good at code and abstraction and know exactly when to switch from one to the other—that is, which things should be described as data structures and which things would be better off being handled procedurally. Then there are the mass-market developers, who try to use code to solve every problem and aren't particularly abstract in the way they design things. That's why readily re-usable components and object-oriented programming turn out to be so hard. So these are folks who really understand procedural logic and yet are not particularly good at

high-level extraction. Lastly, there are the business analysts and general users who are not programmers but nevertheless are quite good at both abstraction and complexity. These are people who might use Access or Vision to build extremely complex diagrams or extremely complex databases. Anyway, you want all of these people to be able to work together. So I've slightly modified Alan Kay's principle to read as: "Simple things should be declarative while hard things should be procedural."

In terms of how we intend to provide for all this, we foresee a natural partnership between power users and programmers. Our goal is to allow corporate developers to interact with all of the metadata and the XML. And I think languages in general are being driven toward a more native way of supporting that.

KM Are there any other major transitions you foresee over the long haul?

AB The biggest change I see is that we're moving away from a data-centric world to a message-centric world. Throughout the '90s, we witnessed the triumph of client/server computing. We also saw a vast number of changes in programming that made it easier to talk to databases. So a lot of that was about writing data-centric applications. And that's the classic two-tier model. But now we're moving to an n-tier model. And with an n-tier model, the real problems have to do with exposing too many specifics to systems outside your immediate family. Because when you do that, you break.

Over the next 10 years, we're going to move toward communications that are message-oriented, with systems talking to each other through public contracts in asynchronous ways. After that, I think a lot of the changes we'll see will have to do with optimizing that communications scheme. Even today, we have customers asking us to move as many as 500,000 messages a second.

We're also going to have to add procedural intelligence to the way in which messages flow. We can't do that right now. You can move things at the rate of 500,000 a second if you have very highly tuned multicasting. But our customers are asking us to write message-oriented models that deal with how the messages flow, how you control who you send messages to and that sort of thing. And they want all of it to scale. That may sound pretty daunting, but it may not actually be as bad as we think. After all, in the mid-1980s, when it became clear that relational databases were the way to go, Oracle was still a tiny toy in terms of actual performance. But as more and more customers moved in that direction, Oracle and other vendors managed to deliver the performance they had to. And there are many other examples of that as well.

Over and over, history has shown that if you look forward five years from the time a technology first emerges, you almost always end up making less progress than you expected. But if you were to look at a graph of progress some time later, you'd realize that you were just then coming up right about then to an inflection point, such that over the course of ten years, you'll actually have made much more progress than you had expected. For example, we all started trying in earnest to get client/server computing to take off right around 1990, which is when we built the plumbing and all the other

necessary pieces. But it wasn't until 1995 that things really started to take off. In the same manner, over the past two or three years, Web Services have moved slower than anticipated. We haven't witnessed the adoption cycle we had anticipated. But consider that this has all just gotten rolled out over that period. If you look out over 10 years, that's where you'll be able to see the other side of the curve. There will definitely be a point where you'll see the volumes pick up at a rather astonishing rate.

Going back to the database example, ultimately, Oracle bet on a future standard and the power of that standard has carried them through to where they are today. Now, other companies didn't fare nearly as well and that's what I expect we'll also see in the area of Web Services. There are companies today that are betting on certain proprietary standards for messaging and application integration—some of which are quite strong. But when you look at where things are going to be 10 years from now, I think you'll find the landscape will have changed rather dramatically.