Data Routing Rather than Databases: The Meaning of the Next Wave of the Web Revolution to Data Management

Adam Bosworth

BEA Systems, Inc.
2315 North 1st Street
San Jose, CA 95131
U.S.A.
adam.bosworth@bea.com

Abstract

What is going to be as important in the next 20 years as relational databases were in the prior 20 years is the management of self-describing extensible messages. The net is undergoing a profound change as it moves from an entirely pull-oriented model into a push model. This latter model is far more biological in nature with an increasing amount of information flowing asynchronously through the system to form an InformationBus. The key challenges for the next 20 years will be storing, routing, querying, filtering, managing, and interacting with this bus in a manner that doesn't lead to total systems degradation. Predictive intelligent filtering and rules engines will become more important than querying. Driving factors for this revolution will be the need for push for portable devices due to their poor latency and intermittent communication, an increasing demand for timely information on fully connected devices, a huge rise in application to application integration through asynchronous messaging based on web services and a concomitant requirement for an entirely new type of message broker, and an increasing desire for intelligent agents to cope with information overload as all information becomes available all the time. The key enabling technology will be XML messages and the various technologies that will develop for handling XML ranging from transformation to compression to indexing to storage to programming languages.

Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002

Introduction

These are the ruminations of an old man who for twenty-five years has seen a lot of change and had a share in causing it. They are simplistic because my experience has been, over and over again, that the winning ideas are essentially simple in conception.

First I have a question. How many of you remember Cullinet? How many of you know Oracle? Twenty years ago, the answers would have been very different. What happened? We reached an inflection point that required standards. Oracle got it. Cullinet didn't.

Part One: Change is Coming

History has shown that a new technology is a necessary but not sufficient cause for a new industry. It requires, in addition, one or more disruptive innovations that offer the potential of exploiting this new technology in a manner that offers huge value to customers. Thus the chip itself wasn't sufficient. The critical disruptive innovation was the idea of the PC, which offered automation without the costs and delays inherent in the mainframe driven MIS departments. Similarly, Ethernet wasn't sufficient. What was required was the idea of the LAN (Sun's famous dictum that the Network is the Computer), which offered the ability to share information between programs and people. File sharing wasn't sufficient. The disruptive innovation that really led to explosive growth for Sun and the Lan's was client-server computing which offered shared access to scalable transacted information to literally unlimited numbers of programs residing elsewhere on the LAN. It was this change in technology, the rapid adoption of Ethernet and then the LAN that led to the explosion of databases that really has populated this conference. Obviously databases preceded the LAN, but until it the need for standards wasn't critical enough to drive the SQL/ODBC/JDBC revolution, which in turn fed the explosive use of data.

Each disruptive innovation has driven an entire new wave of computing. The 60's saw the rise of the mainframe driven by the transistor technology of the 50's using the new ideas of COBOL and Fortran. The 70's saw the rise of the minicomputer driven by the integrated circuit/chip technology of the 60's and networking and next generation computing languages. The 80's saw the rise of the PC's driven by the next chips, RAM, and increasingly DOS and C as a standard. The 90's saw two revolutions happen both driven by TCP/IP. In the first half of the 90's we saw the rapid emergence of the LAN driven by the merging standard of TCP/IP and then it's corollary, client-server computing, driven by the LAN and enabled by the emerging standards of SQL and then ODBC. In the second half of the 90's we saw the rapid emergence of the Internet and its corollary, lightweight clients, driven by the emerging value of universal access to information and enabled by the emerging standards of HTML and HTTP.

Each wave has increased the number of users by almost an order of magnitude. As I said, in each case the technology alone wasn't sufficient. An idea, a disruptive innovation, suddenly suggested uses of this technology that were of huge value to a great number of people. But history also shows that only with the rise of standards can these ideas truly drive the explosive growth that they warrant. Consider. The Arpanet has been around for 30 years. Why did the Net only take off in the 90's? It only took off then because the standards of TBL, HTML and HTTP, were critical standards that made possible the development of tools and runtimes to exploit it. Similarly the idea of databases has been around for 40 years or more. Why didn't LAN's take off in the eighties? 3COM was there throughout the eighties. It required the acceptance of TCP/IP to really make it easy for everyone to use LAN's. Why didn't the database community really take off until the late 80's or early 90's (or put differently, why has everyone heard of Oracle and no one heard of Cullinet)? Databases only really took off when the standards of SQL and TCP/IP and a bit later ODBC could be counted on to make possible the widespread development of the tools and runtimes that we take for granted today. Put more bluntly, Oracle in 1985 wasn't a VLDB. It became one because it had to because of the demand that these standards and their companions, the emerging tools and applications put on it. This group owes its success to the standards of the LAN and the standards of SQL and VLDB.

The Internet enables the next wave, which in many ways is really just beginning. The Internet is a communication device. As such, it enables anyone or any application to access anyone or any application at almost anytime. We have already seen the scope and impact of the first applications, person to person (email and I/M) and person to application (HTTP/HTML). These applications have driven fundamentally new computing paradigms organized around massively scalable architectures with stateless programming models designed to handle the unpredictable load that universal access implies. We now are moving into the opportunities that applications invoking other foreign and remote applications imply. These opportunities alone would be as massive as those created by people invoking applications. They imply the true sharing of processes and information on a new scale. They are enabled by newly emergent standards of SOAP and XML and WSDL, namely Web-Services. But they are not alone. We have recently moved into a world of wireless communications. How many in this audience do not have cell phones? Indeed how many do not have some form of wireless email or calendar? The new technologies of wireless communication turn out to require very similar architecture to that of applications integration. The purpose of this talk is to discuss what that architecture is.

Most of the applications we have built to date have had an architecture centered on pull. An application to display a list of tasks and let me edit it will pull these tasks from the database, typically using a query to do so. True, it will push the changes back using a query, but the client side of the application, which is where the "application logic" resides, is pull. This is unchanged even by so fundamental a change as web browsing. Indeed, pull is a more constant feature in this world because the lifetime of pages is short, virtually evanescent in a well-designed system, and that means that almost all access to data is a quick pull. You cannot push information into items whose lifetime is measured in milliseconds. Even in this world, however, there has been another side to the equation. If the result of a page is to request some work to be done, for example inserting a new purchase order or cancelling an order, then typically the request is queued. This means that there is code, somewhere, that is servicing this queue and making the requisite changes to the database. Oddly, databases have never taken queues that seriously even when, as with Oracle, they support them. This is going to change.

When applications talk to each other, they are typically going to use push instead of pull. In other words, rather than sending a request and waiting patiently for it to respond, they are going to send a message (push a message to the other application) and service queues that the other application will use to send messages back. Only to a lesser degree do I think they will synchronously invoke each other. Why so? Well, there are 3 fundamental reasons for this:

1. Applications are not always available 24 by 7. Almost all applications are down some of the time. Some seem to be down much of the time. Some of the time, they aren't down, but they are executing batch processes and aren't prepared to provide

interactive services. Ideally, a program will not hold the memory for a thread and block for any significant period of time, which is what the requesting applications would have to do in this case. It becomes a single risk point for failure. When people are waiting and the browser is the only game in town, there is no choice. But when applications are waiting, they are mindless and will often patiently retry over and over again every 10 seconds which just queues up more and more requesting threads to the unresponsive application. Bad idea.

2. Applications weren't written for unpredictable load, but enabling any application on the net (or on devices) to invoke them invites exactly that. Most applications were written assuming some limited number of humans within some enterprise would invoke them. They can only scale so far. If the load surges during peak periods and each request is a synchronous request/response, either the server will deny the request (which means that the requesting app will either fail or retry which will make things worse) or it will try to handle more requests than is good for it and will bog down and ultimately start to thrash. We saw this in the early days of the web. It was called firestorms or web-storms and typically resulted from naive programmers queuing up more threads as more demands came in rather than recognizing that there was/is a native limit to the throughput of a processor. There are also QOS issues involved. Some requests matter more than others. EBay for example, wants to service requests to add goods to an auction at a much higher priority than requests to return the results of a query about what goods are available at a given price/type. Trading systems want important clients to get faster feedback than nickel and dime ones. There are not good mechanisms for doing this if all invocations result in synchronous threads in the serving application. There are if all invocations result in queued requests. It is called intelligent de-queuing.
3. Last, but certainly not least, most applications in the real world involve people, process, and time. Ask a bank for my credit history. A person in a workflow may have to authenticate and approve this request. Ask a paper-mill to make you paper and to let you know when the paper is ready and weeks may elapse. Ask a financial application to compute a hedging strategy and an analyst may have to work it out by hand. Even ask a database a sufficiently hard and/or expensive question (say some assay of oil drilling results or genome analysis) and the answer may be slow in coming. All of these delays are best handled, if a robust reliable scalable architecture is desired, with messaging. Then the programming model fits the reality. It understands that long periods of time may elapse between the request and the response (or responses for things where status messages are expected).

In some sense there is no news in this point. Those companies that have mastered the arcane minutia of applications integration heretofore, SeeBeyond, Vitria, Tibco, WebMethods, and so on have all understood this and built as their core competency a messaging bus that, plugged into the applications by suitable "adaptors" handles a world in which much of the interactions are asynchronous. But these times are driving a new and profound new importance to messaging. Just as the advent of the LAN caused the marginal benefit of shared data across applications skyrocket and so caused the benefit of shared standards skyrocket and so drove the client server revolution that pays most of your salaries today, so the advent of the Internet has caused the marginal benefit of applications integration skyrocket and so driven the benefit of shared standards (Web Services and much more that is missing, workflow, message filtering and routing, content-driven pub/sub, conversations) to skyrocket and this will in turn sweep over the applications world and leave in its wake a completely new applications model centered on new standards and with new and very difficult problems to solve.

There will be another forcing factor driving this new programming model. This is the rise of wireless devices. Declaring wireless applications dead because of WAP's failure is like declaring the death Internet based applications because of Prodigy's failure. Almost all successful revolutions have some early misfires. There are going to be an order of magnitude more people using mobile devices than PC's, especially here in China and in India and the third world where PC's are still far beyond most people's means. But what made WAP fail and SMS succeed? In part, to be honest, pricing. But the bigger issue was that in the real world, wireless communications is unreliable and latency-challenged. In short, you often lose the connection and even when it is there, it can be very slow. A user interface paradigm like the browser (which WAP slavishly copied) which serves up UI in little chunks each of which requires a synchronous round trip to the server before you can execute the next step is not suited to this at all. SMS is wonderfully suited. Sometimes the messages show up instantly, sometimes slowly, but there is no sense of the user being blocked in his/her task. The industry is figuring this out and BEA is pulling together a consortium of mobile players to drive a new next-generation asynchronous browsing paradigm, which is perfectly suited to these realities. In so doing, it will model all communications to and from the server as messages because, again, of the inherent risk of delays. This model also has substantial advantages for the Internet as a whole because a model based on messaging can push new data to the user such as machine problems in the factory flow or emergency requests for approval or alerts that some stock has just dropped below some floor.

The volumes we can expect in this arena will be nothing short of staggering. Even today SMS messages in Europe move in huge numbers and statistically it is still largely a phenomenon of the 14-24 set. When several hundred million people get used to using their devices as intelligent listeners for information (stock quotes, traffic tie-ups, plant problems, new contacts, shopping lists, project and task changes, contacts, customer orders, school homework and so on) we can start to assume that really large numbers of messages a day will flow through this system, numbers that dwarf the numbers of airline reservation systems or ATM's today. Why? This is true because unlike the airline or ATM scenario the users can be passive. Set up the application once and it will send you messages forever. Often, you will send messages in response, but even then the effort will be minimal since your portable device will always be right there and so you will send more messages. Since I started using a Blackberry for example, the amount of mail I send a day has actually gone up about 30% and the amount I receive has gone up as well because I respond so promptly.

In short, messaging is coming and in huge volumes. So, what are the requisite pieces that we really need in our architecture and standards to make this model work well? And what are the challenges?

Part Two: What is Required?

For this part of the talk I'm going to make an assumption. I'm going to assume that over time most messages will evolve to be, logically speaking, XML. They make be some efficient tokenized compressed form of XML, but it will be isomorphic to XML. I assume this for the same reason that your databases have a standardized language for schema. Messages will need to be self-describing and in a standardized manner. The systems processing them will not be local to the source of the messages and may not even know who sent them. Often the messages will have flowed through a network of processes each of which may have extended the message. XML is well suited to all this. I have met a single customer in 100 discussions this year who disagrees. They worry, a lot, about the performance, but they assume XML. There is simply no other reasonable choice.

So what are the requirements? At first blush it seems simple. You build queues. You add some lock types for read-through so that GetNext can get the next available message in the queue, you ensure that the queues don't bottleneck on queuing operations because of locks in some page and you're done. That was easy.

Smart Queues:

Not so. Because of Quality of Service considerations, you need intelligent filtered ways to pull messages off of the queue. Since these messages are going to control XML (assumption!), then the content will necessarily need to index intelligently on the XML in some fashion to enable fast efficient removal of items.

Message Broker:

But this is only the lowest layer of the plumbing. It turns out that it isn't a good idea to build all connections between applications in a point-to-point fashion. Customers and companies that service the integration space learned this some time ago. If you try and connect n applications to each other in a point-to-point manner, you end up with something like N*(N-1)/2 pieces of connecting code. When you consider that most big companies have literally 1000's of deployed applications, it quickly becomes clear that this is a bad idea. Instead, what is required is something that is commonly called a message broker. What is a message broker? Essentially it is a very smart highly optimized pub/sub server that on receipt of any new message can figure out very efficiently which applications want to know about this message and how to deliver it to them. These are mission critical applications we're talking about and the loads are often millions or tens of millions a day so the need to scale over clusters of processors is paramount. Because the volumes are so high, it isn't always possible to use a classic database as a backing store for the queues. We have many customers asking for 20MM/day now with anticipated load going much higher. That implies peak load of at least 3.6MM/Hour or 1000/second. How do you handle in a single millisecond analyzing/parsing the message, discovering who wants to know about this message, transforming and shaping the message into the one they are expecting, and rerouting it to them? You may not and hence you cluster across a set of processors but then you have other hard issues. What if the order of delivery must match the order in which messages have been queued (often the case)? Clearly the same sorts of intelligent optimization which years of experience have built up in the database community are going to be required here. The languages and standards that describe message filtering and transformations are going to have to be susceptible to a high degree of optimization. It is going to be critical to cache and optimize the compiled rules about who wants to know what. There are new problems. Often if the messages aren't delivered in time you might as well throw them away. There is another message coming with later data and you missed the boat. But how do you have time to throw them away? So ageing and scavenging becomes a new and important task. To be honest, I think this is an area that databases haven't given the attention it deserves anyway, but messaging will force this issue.

Workflow:

OK, now the hard part starts. So far, we have assumed a completely stateless model. In comes a message and the instructions in the message broker looking solely at the message decide who needs to know and how to let them know. But in practice, as with the web, this is often going to be stateful. The developer will send a message asking for a risk-analysis calculation. The same code will have a set of state garnered from earlier messages that responded to requests for information about the customer, the deal, the beta for this company, and so on. Now when the risk system returns its estimate, the code wants to pick up and continue with this information and probably send yet another message letting the original invoker know that this investment isn't a good idea. This requires new programming paradigms for workflow. Notice that it is the workflow itself that interacts with (is hosted by) the message broker to let it know what messages it wants to send and which ones it is waiting for. The workflow may require the illusion that the messages have been delivered in a specific order even though in practice they have arrived in a non-deterministic sequence because this makes the programming so much easier.

Errors will occur. In this world though, for reasons already adumbrated, the normal model for communication will be messaging. So if an error occurs, there will be no practical way to unwind inside of a two-phase commit. The delays between messages will absolutely preclude it and, in the real world, applications do not enlist in external two-phase transaction coordinators anyway. So instead of the reliable ACID transaction model that has guided us so well so far, we will need a model for unwinding, for long-running transactions. Today, despite the use of the word transaction, the so-called long-running transactions are anything but ACID. Workflows and Services will need a formal model for understanding how to back out operations, what state is required, and what the escalation policy if the back out fails should be.

The other hard issue for workflow is going to be management. As described, the lifetime of these workflows can be very long indeed. Customers (and applications) will have expectations about how rapidly they should come to closure. Customers will need to understand what are the bottlenecks, when the workflows are essentially never going to be done (either because there is an inherent circularity in the system or because it is patiently waiting for some message that will never show up). Since these workflows will often interacting with other workflows in other organizations, if one is aborted, how are all the others that exist solely for it garbage collected? This is going to be a hard and thorny problem and I expect that we'll need standard models of Leasing and automatic termination protocols to make it all work well.

Let's assume that the workflow has exported this knowledge into the message broker so that the broker only routes messages to it in the right order. How is this described? How can the code be independent of the implementation of the message broker? All this can only if, as with SQL, the workflow nature of the code is exported in a standard way into the message broker so that any broker can support it. The workflow may only register a desire to be informed about messages with certain content so that the way it expresses the related filtering of messages needs to be equally standardized. Oh yes, all this has to be happening at volumes of millions a day. Or more. So what will this "language" look like?

Workflow Language:

Alan Kay is supposed to have said that simple things should be simple and hard things should be possible. It has been my experience over 25 years of software development that for most software products, simple things should be declarative and/or visual and hard things should be procedural. Declarative languages have an unfortunate tendency to metastasize because people need to do things that are hard. When they grow in this way, not only can most people not use these new features, they find the entire language more daunting. A good example of this is XSLT. It started out as 2 simple visions by 2 groups. We (at Microsoft) wanted an XML Transform language that would typically take input documents with well-known schema and transform them into other documents also typically with well-known schema. Others (Sharon Adler, James Clark, ...) wanted a language that would take semi-structured unpredictable input schemas and generate rich output UI. What is more, some things that we all wanted to do were hard. The result was an overly complex language all because of the desire to avoid code. Code is good. All great products have been procedurally extensible. In your own worlds, you all learned (or were sometimes force-fed) the importance of being able to extend SQL with code whether it was PL SQL or Transact SQL or SQL J. Why would workflow be any different? There will be simple things programs need to do. They will need to wait for certain sequences of messages. They will want to run some things after each message but have others wait until the jury is in. All this encourages a declarative model. It can be handled visually and optimized and is easy to understand. But customers will also want to do hard things. They will want to iterate over elements of an incoming message and run, recursively, the same logic as the overall workflow. They will want certain incoming messages to trap and force exits out of their normal logic. They will want to make intelligent decisions about what to do that simply are hard to describe without Turing complete logic. History has shown that the best way to provide this is to make the language procedurally extensible. This is not a simple or trivial problem. There tend to be three approaches to procedurally extending a declarative language:

1. Write procedural components and reference them from the declarative language. This is the solution that most workflows today use. This is the solution that XSLT uses. It is the solution that SQL uses when it enables callouts to procedural logic. The normal problem with this model is that there are some hard things that still aren't possible because the code

would have to change the declarative logic itself and that the type systems and error handling are often fairly incompatible.

2. Extend the declarative language to be procedural. This is the solution that Transact SQL uses and PL/SQL uses and DHTML uses and Daniela Florescu has suggested for XL. This tends to be tricky in that there are two name spaces, one for the variables in the declarative logic and one for the elements of the data being processed and that often the visual tools that work on the declarative language break down when it is procedurally extended.
3. Extend a procedural language using either new command structures or meta-data to encompass the declarative model. To some degree this is the solution that SQLJ uses and it is the solution that our latest product WLW uses and in a cleaner manner it is the solution that Collaxa uses for workflow. BEA has drafted a proposal for an extension to Java to handle the pi-calculus logic of workflow, which we called Pi-Java but it sure isn't easy to change Java. The language must have set a world record for sclerosis. This model has the advantage that the procedural language is already well known to programmers and works well procedurally, but often has both the problems with both data-types and the problems with visual editors.

In either case, if workflow is to follow this dictum, it will start to include code. In short workflows will sometimes be programs. These programs will be processing XML messages. They will frequently be splicing out interesting pieces and saving them in anticipation of subsequent messages. They will often be constructing new XML messages from parts of this state. They will make decisions based on the values. In, short XML is going to need to be a native data-type for these programs. We, at BEA, have started experimenting in this arena by joining in an effort with others such as Netscape to extend Ecmascript (Javascript) to incorporate XML as a native type and to open source this work. We collectively submitted this work to the EcmaScript community and it has voted, unanimously to incorporate these additions into its next version. We are in the process of open-sourcing the work we've done. It has been interesting and turns out to be quite difficult to get right. This is one possible direction. At the same time Donald Kossman and Daniela Florescu and others funded by working on a project funded by BEA are experimenting with an extension to XML Query language, which adds in procedural logic and workflow primitives. This is also a promising direction. Which will it be? I truly don't know. What I do know is that it will not be some extraordinarily over-designed declarative language whose complexity can be measured by a spec inches thick. Remember. Simple things should be declarative. Complex things should be procedural.

This really matters. This is, essentially, the SQL of this world. It is the descriptions given to the message bus.

So let's review. The message bus will need to process 1000's of messages a second in a reliable transacted way where the logic associated with each incoming message may well be procedurally extended and the very decision about which logic needs to be run may well itself be procedurally extended using some language that is comfortable with XML. My friends, this is called an application. The application logic is going to move, to a very significant degree, into the message broker itself. But at the same time, in order to achieve the performance we need, simple things are going to be to be computable into highly efficient execution schemes.

Daunted? But wait, there's more.

Intelligent XML Databases:

The raw material for all of this processing isn't going to be characters and simple data types as was true with the old modal programming models of the 60's and 70's. It isn't even going to be mouse moves and complex types as became true in the 80's and 90's as the work of Xerox Park in the 70's and Simula in the 60's finally percolated into the mainstream. It is going to be contexts and complex XML messages. Thus the state that the language will need to access will frequently be XML itself and over and over the programs and workflows will be pulling out pieces of XML. Often this XML will need to be stored and retrieved between messages. Often the workflows and programs and rules will be pulling out complex pieces of it much as the biological community today pulls out pieces of DNA. They will be looking for patterns in the incoming stream of XML again much as the biological community today looks for patterns. If there is an area that will probably need to inform us about this model, it is probably Bio-Informatics. As a simple example, a workflow may want to run some exception processing logic if an incoming message has a history showing that it has already been rejected twice by other services. But how does it know? The workflow will need to describe patterns to search for. Today, our stores are quite poorly optimized for all this. In this world, we need persistence across invocations, we need highly intelligent and efficient pattern driven retrieval of pieces of the message, and we often are willing to build in machine affinity routing logic to be able to keep information in memory and avoid the overheads of multi-user locking in order to get the performance and scale that we need. The data services that will need to underlie the business logic described by the procedurally extensible workflow will need to be able to provide very fast and optimized plans for pulling out pieces while at the same time provide very fast ways to serve up the entire message at once or splice together new messages from fragments of old ones. Will this database push its data to intelligent rules engines rather than wait for queries?

Summary

So, let me wrap up. I stand in front on one of the smartest groups of people on the planet. All of you have far more degrees than I do, keener minds, and a desire to push the envelope of knowledge and understanding. It is time for all of you to turn your attention from the well-understood and increasingly static area of scaling relational databases into the critical and exciting Mount Everest's that now need to be surmounted. From a simple seed, asynchrony, a huge forest must be grown. None of it will be easy and the problems will be as complex as those in database. As with the relational database world, it will take 10 years to come to full fruition. It will have its own mathematically logic between the predicate calculus and the pi-calculus. What it needs is the experts and new languages and new algorithms that enable it to grow to meet the urgent customer needs because, with the advent of the Internet and the wireless device, the time is now.

Biographical Sketch

As VP of Engineering for BEA's Framework Division, Adam Bosworth drives the strategic and technical directions for BEA's WebLogic Workshop, WebLogic Integration, and WebLogic Portal products. Before joining BEA, he co-founded Crossgain, a software development firm acquired by BEA in 2001. Adam Bosworth is widely recognized as a pioneer and key figure in the evolution of XML. Prior to Crossgain, he was a senior manager at Microsoft where he drove the company's entire XML program from 1997 through 1999. He was then named general manager of Microsoft's WebData organization, a team focused on refining the company's long-term XML strategy. While at Microsoft, he was also responsible for designing and delivering the Microsoft Access PC Database product, and he managed the development of the HTML engine used in Internet Explorer 4 and Internet Explorer 5.