Abstract for HPTS
Don't come to us, we'll come to you
Adam Bosworth
Chief Architect, BEA

Overview:

In the next five years, we'll see five fundamental changes occur in enterprise computing:

1) Enterprise computing will evolve into a Service Oriented Architecture
2) Most new applications will become heavily driven by XML based meta-data that can be dynamically modified by administrators and even business users to tune and modify running applications
3) Most applications will be loosely coupled to others using them to integrate at the user interface level, the data level, and the process level using XML as a messaging format for the loose coupling
4) Due to the promises made about a Service Oriented Architecture, customers will increasingly try to get access to real-time information or at least close to it
5) Far more information will start to flow through the system. RFID, mobile computing with GPS sensors and other such advances will generate huge amounts of data. To put this into perspective Rob Carter, CIO of Federal Express, says that his company generates 60 terabytes of data every 180 days just from the transactions tracking the goods Federal Express ships and that most of this information is going to flow in XML formats.

All these factors will change how we think of integrating access to information. The problem will not be distributed SQL queries. It will be distributed queries over applications and over interim stores of XML. Nor will it be entirely pull based as is the case today. Instead, the information will flow to the listeners. Push, not pull, will be the problem we deal with during this decade. XML, not rows in tables will be the critical raw material.

Justification:

Even in the last five years, we've seen a sea change in this direction.

Consider the world of meta-data. Virtually all meta-data for applications is now described in XML. This includes security policy, the deployment information for components, the description of portals and portlets, the contract (WSDL) for loose coupling of applications, the description of workflow and choreography, and much more. This collection of information is normally known as a "repository". As this data changes, the changes must flow into the running applications.

Consider the world of Enterprise Application Integration. We've seen customers come to realize that the classic model of data integration based on distributed queries, data ware-housing, and ETL's isn't sufficient for them because their data is increasingly protected by fortresses known as applications and cannot be directly queried. As these applications

evolve into reusable services this doesn't resolve this issue. Indeed, the need to *loosely couple* to these services to ensure that change doesn't break the architecture only strengthens the need to not directly talk to their databases. Thus customers are starting to take a long hard look at integration technologies such as Liquid Data which abstract all data sources into XML based services and distribute queries across them. Their primary request is to give them the ability to dynamically flow changes from applications into this unified data model.

Consider the world of business process and workflow. As business processes integrate services via the mechanism of web services, it has become clear that the basic building blocks that they consume, persist, extract, and manufacture are XML ones. Thus the transient storage model used for in-process business processes is increasingly an XML based one. The relational database isn't up to this task. Instead it is used as a transacted indexed file system storing blobs. In short it is used just to reliably checkpoint the information and serve it up on demand. Obviously, this information flows in every time a new message for a running business process arrives.

Consider the back end processing in most major enterprises. The back end processing essentially is listening to information published and queued by a heterogeneous sea of data capture and front end applications. The number of initiating messages can be huge, tens of millions a day and each initiating message can move through a plethora of processes. It has become natural to use XML to record both the initiating transaction and the updates made to it in the course of processing because the web service paradigm is driving this as an exchange model in either case.

Consider that most applications *aren't* designed to be queried in general purpose ways. Thus implementing distributed query engines that query them isn't sufficient. It is far easier for the applications to "publish" all new information as it occurs than for them to support general purpose queries. This model can coexist well with a distributed query system which materializes views and then "maintains them" by subscribing to these change messages as long as it is understood that the "consistency" model is far from perfect.

Consider that the key push in the financial industry is one in which the settlement should be moved from days to hours or even less. This is accomplished, throughout the financial industry not by some distributed query model but by smart switches listening to all trades and matching them up in complex ways and pushing the changes to interested systems and applications.

Consider the problems of scale. A lot of messages are moving through the back end. The volumes here can be daunting. There are customers who are generating well over 100,000,000 transactions a day. That's about 6,000,000 an hour or about 2,000 a second. This in turn tends to argue for fully distributed shared nothing architecture or, in other words, a cluster. As far as our customers are concerned data bases do not make this easy. They view the database as the only limitation in their system to dynamic limitless scale. While Oracle and DB2 enable partitioned storage across machines, the customers tell us

that they really aren't designed to dynamically and reliably scale across a cloud of cheap machines in the way that a cluster is. Customers are quickly seeing the price point differences between the cheap stock machines that they need to scale their "front end" web servers or even their application servers and the huge prices that they pay Oracle and IBM for massively scalable data-stores. They are looking for similar economies of scale and reuse of cheap stock components in this arena and not finding it.

Consider the problems of change. The business process is changing all the time. BEA's customers tend to modify deployed applications an average of once every six weeks. Their real dread is when they need to change the database schema. Often they start packaging the extensibility inside of fields that are opaque to the database. We call these fields XML. Then they can alter and extend the schema at will, instance by instance, if need be. If they don't do this, then the database becomes a bottleneck to their systems in more ways than one, it limits the throughput compared to the arbitrary scaling of a cluster and it limits their ability to change compare to the relatively effortless deployment of a new JSP.

Challenges:

These new technologies have some of the classic problems. How do you query? How do you update? How do you ensure consistency? How do you scale? How do you guarantee a quality of service? How to you dynamically modify and alter running systems while maintaining five nine's?

The questions of scale and service and change become paramount in a Service Oriented Architecture. There are really only two solutions we have found in the industry for scale:

1) A shared nothing architecture currently known as a cluster in which a set of machines can dynamically process requests as they arrive in the system
2) Caching in which data is allowed to be stale using what is known as TTL (or time to live) and other forms of leasing so that when resources do have to shared they can be shared less frequently

Accordingly, as customers roll out the new XML based solutions for meta-data they are increasingly copying meta-data to each machine in the cluster and caching it there. Similarly, as the new XML based data integration technologies roll out again customers are increasingly copying XML to each machine in the cluster and caching it there. Already many of them are building classic hierarchal managers to manage these distributed caches so that they can invalidate the cache when necessary.

In both cases, this opens up the following questions:

1) How do these caches get updated when the data that they are caching changes?
2) If we do want to "push" changes to the machines that are caching this information, how do we know which machines in the cluster need what information?

3) If the changes need to be made atomically and consistently and yet cannot require a call back to a central resource provider at all times because of performance requirements (for example security policy), then how do we make such changes?
4) If the user wants to "write through" the cache back to the shared providers of the information, how do we know who to write back to and in what format?
5) How do we query the local cache of information?
6) If the information being cached is large, then there is the additional requirement to have a disk-based model for the local cache and the query engine must be able to handle large sets.

In essence, in each case, the machines in the cluster are "subscribing" to one or more services that have the critical data that they need. This is equivalent to the problem of maintaining materialized views with the following additional constraints:

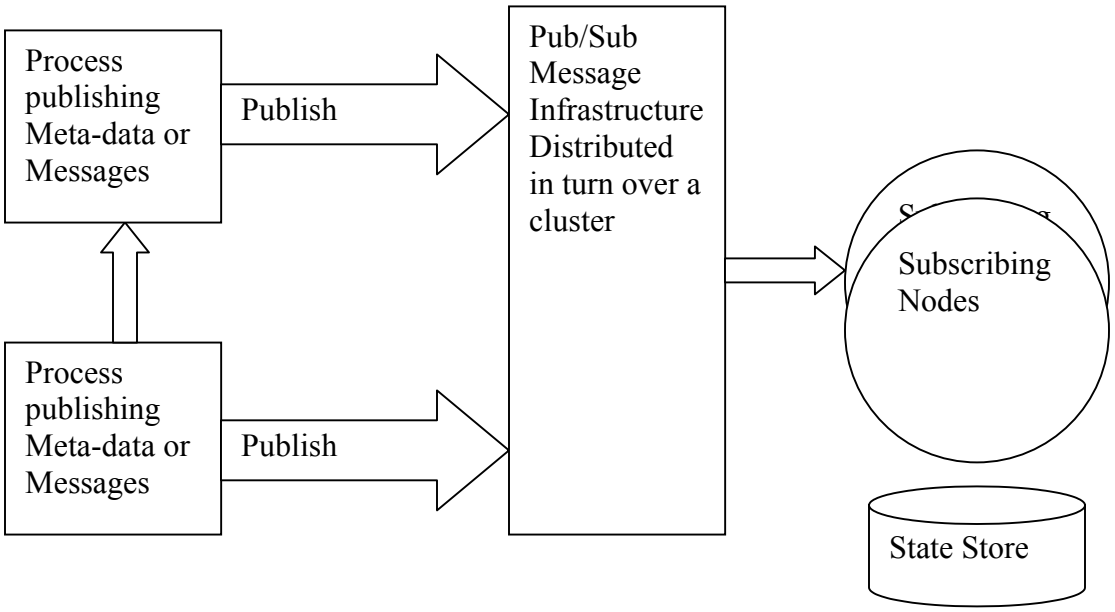
1) The provider of the resource doesn't control the machine(s) that are materializing the view
2) There is no guarantee in a transactional sense that the view is consistent
3) The manipulations of the data are much richer than normal view semantics and are *not* understood by the original publisher of the information. For example, PeopleSoft may provide information about an employee and Siebel about the Employee's customers and the "join" across the two may involve some complex procedural or workflow logic that is the "service" that unifies the two. In this case, a downstream consumer can only subscribe to the joining service, but even then, updating back through this service becomes much more problematic.
4) There will normally be cache TTL semantics which is dramatically different from a materialized view world.

All of this requires some new ways to think about data and queries. Instead of using SQL, we believe that XML Query becomes the suitable language for these areas because of the pervasive use of XML due both to the complexity of meta-data and the need to loosely couple interactions between computers.

Secondly, we think this requires new ways to think about XML and updates. In order to know what to update in an XML "cache" it is important that the updatable items have a unique ID and version number. This is much like the version number/timestamps and primary keys in a relational database except that XML is a graph and the only primitive in XML is an element and it isn't true that all elements will need this overhead, only those that can repeat.

Third, this requires a new engine for publish and subscribe which we call a message router which can act as outside of the underlying applications because it is extremely unlikely that the underlying applications will be smart enough to have their own inverse filter/query engine. Instead, they will simply publish new messages any time there is a change and it will be up to a "message router" to route them to the appropriate subscribers. This is true because, unlike a database, there is no central manager for applications making sure that the semantics of all queries is understood. Today, customers roll their own form of these routers and call them message switches, ticker

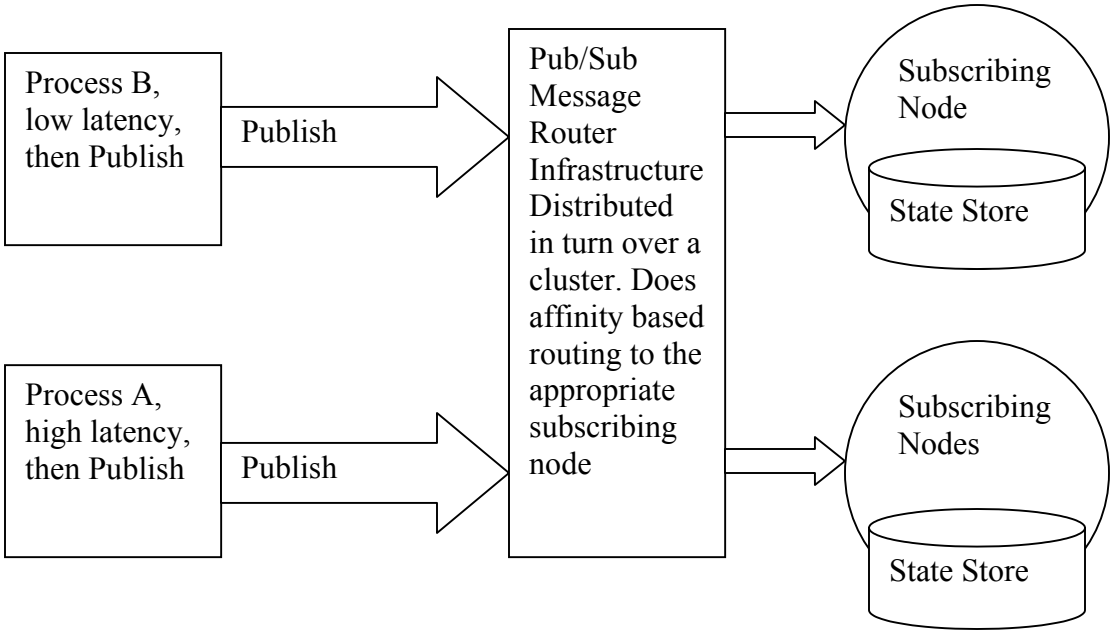
plants, and a myriad of other names. They are programmed by hand by administrators and no one really understands well if the resulting listeners know what they need to know when they need to know it.



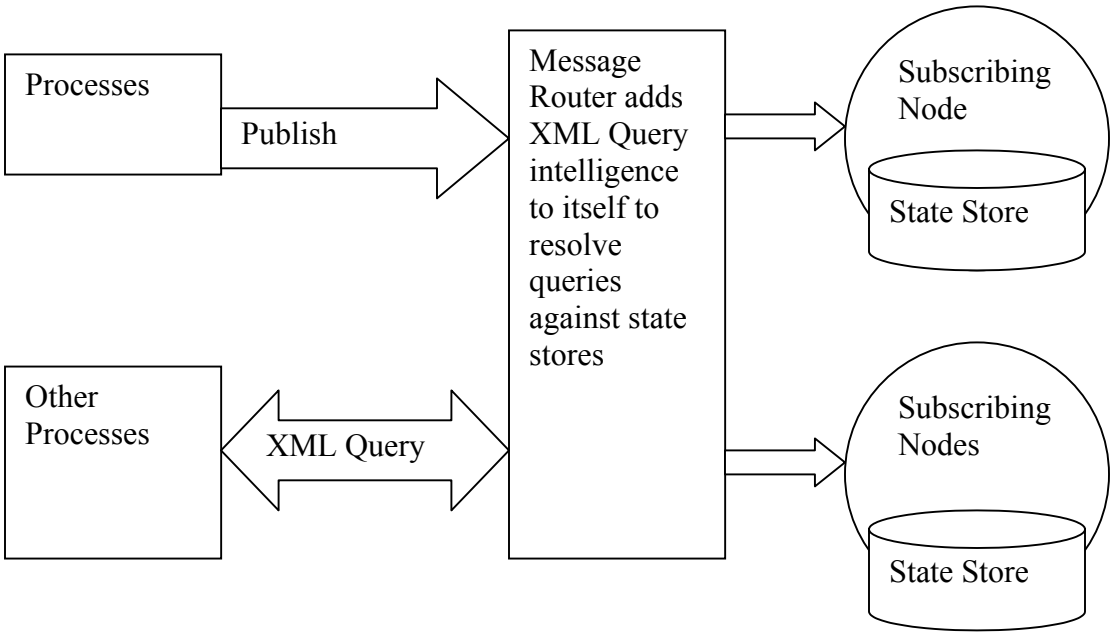
The volumes are already significant. We have customers who are generating well over 100,000,000 transactions a day. That's about 6,000,000 an hour or about 2,000 a second and at peak times often higher. This in turn tends to argue for fully distributed shared nothing architecture or, in other words, a cluster. However, the processing of the messages often needs to be stateful and the machines in the cluster cannot provide stateful behavior for a sequence of related messages unless they, and only they, receive the messages *or* all processing is checked to and from a shared database. Our customers are asking us to move to the former model to avoid the perceived "bottleneck" in latency and throughput of a database.

To handle retrieve queries in these volumes is also problematic, limited in headroom, and to say the least, hugely expensive. If the messages are XML based as was posited earlier, then there is the issue of storing and retrieving them in a database. Customers tell us that classic object relational models do not work under such loads. If they are stored as blobs, then the value of the database as an engine for queries is sharply reduced and it becomes reasonable to ask why not just use some form of affinity routing and a partitioned store that is optimized for caching, indexing, transacted storage, and linear scale. SleepyCat, for example, might be sufficient and obviously, a heck of a lot cheaper.

Then the picture above is modified as follows:



From here it is natural step to think about the following picture:



In this picture, there is a distributed data store for messages and meta-data. It can be queried, but completely scalable. It isn't intended to replace existing databases. It is intended to provide a model for efficient and massively scalable access to the core plumbing for the next generation of systems for which, meta-data and in-flight state become core information. Obviously building this is a major endeavor. It is our belief that this is the architecture required to meet our customer's needs in the era of Service Oriented Architecture.