Learning from **THE WEB**

ADAM BOSWORTH, GOOGLE

W

In the past decade we have seen a revolution in computing that transcends anything seen to date in terms of scope and reach, but also in terms of how we think about what makes up “good” and “bad” computing. The Web taught us several unintuitive lessons:

1. Simple, relaxed, sloppily extensible text formats and protocols often work better than complex and efficient binary ones. Because there are no barriers to entry, these are ideal. A bottom-up initiative can quickly form around them and reach a tipping point in terms of adoption. In other words, anyone can write HTML, no matter how syntax-challenged they may be, because the browsers are so forgiving; and even writing an HTTP server is within the reach of orders of magnitude more than, say, writing a CORBA or DCOM server. What’s more, if the text format doesn’t work, one can easily mail around the HTTP request or HTML to friends who will examine it in

The Web has taught us many lessons about distributed computing, but some of the most important ones have yet to fully take hold.

Learning from THE WEB

their text tool of choice and explain what is incorrect. In short, having a format that “normal” people can inspect, understand, augment, and author is hugely important to adoption in a bottom-up world.

2. It is worth making things simple enough that one can harness Moore’s law in parallel. This means that a system can scale more or less linearly both to handle the volume of the requests coming in and, ideally, to handle the complexity. For example, Google is able to handle huge numbers of requests that ask for quite complex work (find all the documents out of billions that are “most relevant” to the following words) largely because it can scale linearly with respect to both the size of the underlying corpus and the volume of queries. DNS is, of course, the best example of this, but so is caching, which brings me to point 3.

3. It is acceptable to be stale much of the time. Most data isn’t updated frequently, if at all. It is inserted as a new version (because of audit trail issues, etc.). Therefore, it doesn’t matter if what you see is a bit out of date. For example, if one uses the tracker for FedEx to look for a package and the data it shows is 30 minutes out of date, it will hardly be catastrophic. Unless one runs too fine a line on meetings or catching flights, even an airline’s landing info may be two to three minutes out of date without any serious ramifications. This fact inherently enables scalability by enabling the lazy copying of data to extra disks.

4. The wisdom of crowds works amazingly well. Successful systems on the Web are bottom-up. They don’t mandate much in a top-down way. Instead, they control themselves through tipping points. For example, Flickr doesn’t tell its users what tags to use for photos. Far from it. Any user can tag any photo with anything (well, I don’t think you can use spaces). But, and this is a key but, Flickr does provide feedback about the most popular tags, and people seeking attention for their photos, or photos that they like, quickly learn to use that lexicon if it makes sense. It turns out to be amazingly stable. Del.icio.us does the same for links (and did it first, actually). Google’s success in making a more relevant search was based on

leveraging the wisdom of crowds (PageRank). RSS 2.0 is taking off because there is a critical mass of people reading it and it is easy to read/write, so people have decided to leverage that when publishing content. It isn’t that it is a good or bad format for things other than syndicated content (for which I think it is very good). Rather, it works well enough.

5. People understand a graph composed of tree-like documents (HTML) related by links (URLs). In some ways I find this the most surprising of all. For years we assumed people had trouble with trees, never mind graphs. And suddenly hyperlinks come along, and as long as there is a Back button, they work.

UNSURPRISING LESSONS

There were some unsurprising lessons to be relearned as well:

6. Pay attention to physics. As a simple rule of thumb, an individual server can normally handle perhaps 100 requests a second (I’ll say within one order of magnitude up if simple ones and down if very hard ones). If the goal is for each server to support 1,000 concurrent users, therefore, try to avoid an event model in which each user pings the server more than once per 10 seconds, let alone one. In short, avoid a fine-grained model of code on the server responding to events at the level of mouse moves or keys typed or each time the scrollbar is dragged three millimeters because if you do, you’ll be generating about 10 events per second or two orders of magnitude too many. By the way, even if supporting only 10 concurrent users per server is acceptable, communications are often fragile, and it isn’t a great idea to be extremely fine-grained because a small rare glitch in communications can much more easily break the system.

7. Be as loosely coupled as possible. Ideally, the server runs asynchronously with respect to the client. They are loosely coupled with respect to time. Then it is easy to maximize overall server throughput and handle prioritization of requests and failover. Unfortunately, this isn’t widely practiced because of the nature of most of today’s applications frameworks, which do not make asynchrony

easy enough for mere mortal programmers, and because the one thing about which asynchrony is unreliable is guaranteed latency, which is essential to a good user experience. The frameworks were derived to support killer apps, which were all about user interface. Failing asynchrony, then at least have a model in which there isn't a hard link (meaning an IP link) between the client and the server because load and code on servers are always altering and machines do fail. So always go through indirection on the way to the server. URLs and redirects both obviously serve this purpose and, between them, do it very well. Some people swizzle at the DNS level and others at the redirect level, but both let one handle moving load and code around, even at runtime. In either case, don't require shared secrets, let alone shared code, to make things work. For example, the browser will continue to work with a site even if the code at the site is completely replaced and the language switched and the operating system switched. There is a limit to this. HTTP and HTML are shared secrets between the browser and the server, so perhaps the rule should be to have incredibly few of them and make them incredibly simple, flexible, forgiving, and generic.

8. KISS. Keep it (the design) simple and stupid. Complex systems tend to fail. They are hard to tune. They tend not to scale as well. They require smarter people to keep the wheels on the road. In short, they are a pain in the you-know-what. Conversely, simple systems tend to be easy to tune and debug and tend to fail less and scale better and are usually easier to operate. This isn't news. As I've argued before, spreadsheets and SQL and PHP all succeeded precisely because they are simple and stupid—and forgiving. Interestingly, standards bodies tend to keep working on standards long after they should have been frozen. They forget these lessons and add a million bells and whistles that would, if adopted, undoubtedly cause the systems to fail. Luckily this doesn't happen because by then there is a large body of installed code (and even hardware) out there that assumes the simpler spec and cannot handle the new bells and whistles. Therefore, no one uses them and we are all protected.

APPLYING THESE LESSONS TO XML

Some of us learned these lessons seven or eight years ago and applied them to distributed computing. We had the explicit goal of using XML over HTTP to exchange information between applications via messages to build a loosely coupled, robust, Web-friendly model for distributed computing. In my humble opinion, however, we ignored or forgot lessons 3, 4, and 5.

more queue: www.acmqueue.com

Lesson 3 tells us that elements in XML with values that are unlikely to change for some known period of time (or where it is acceptable that they are stale for that period of time, such as the title of a book) should be marked to say this. XML has no such model. You can use the HTTP protocol to say this in a very coarse way, but, for example, a book may mostly be invariant, though occasionally have a price change or a new comment, and certainly an item up for auction may see the price change.

Lesson 4 says that we shouldn't over-invest in making schemas universally understood. Certain ones will win when they hit critical mass (as RSS 2.0 has done), and others will not. Some will support several variant but successful strains (as RSS has also done where most readers can read RSS .92, RSS 1.0, RSS 2.0, and Atom). It also suggests that trying to mandate universal URLs for semantics is unlikely to work because, in general, they will not reach a critical mass, and thus people will not know about them—never mind take the trouble to use them. In short, if a large number of people aren't using something or there isn't a feedback mechanism to know what to use, then it will not be used.

Lessons 1 and 5 tell us that XML should be easy to understand without schemas and should let the clients intelligently decide how and when to fetch related infor-

Simple systems tend to be easy to tune and debug and **tend to fail less and scale better** and are usually easier to operate.

Learning from THE WEB

mation, especially large files such as photos, videos, and music, but even just related sets of XML such as comments on a post, reviews of a candidate, or ratings for a restaurant.

Hmm. There are some interesting implications in all of this.

One is that the Semantic Web is in for a lot of heart-break. It has been trying for five years to convince the world to use it. It actually has a point. XML is supposed to be self-describing so that loosely coupled works. If you require a shared secret on both sides, then I'd argue the system isn't loosely coupled, even if the only shared secret is a schema. What's more, XML itself has three serious weaknesses in this regard:

1. It doesn't handle binary data well. No sensible person will encode large amounts of binary data into XML. But then no sensible person would push binary data in the first place. They would let the receiver get it using existing protocols for streaming in binary data (or maybe BitTorrent these days).

2. It doesn't handle links. The world is not one giant tree that can be freeze-dried and put into text. You have to break it up into chunks. And one man's chunk is another man's set of chunks. The way to deal with this is self-describing links, which could, optionally, show you the MIME type and/or purpose of the link as in `<Link purpose="comments" type="Text/Atom"/>` and/or a self-describing syntax for what the server "thinks" is chunks (see point 3 below).

If you examine XML, it typically is made up of *sets* of data and even *sets of sets* of data. The problem is that one cannot look at XML (supposedly self-describing, remember) and know which elements are instances of a set and which are not. If one sees, for example:

```
<Person><Name>...</Name><email>...</email>
<email>...</email></Person>
```

one can't safely assume that there is a set of `<email>` elements. Now the schema will describe this but, it's both hard (the XML schema language is very complex) and,

more to the point, requires a shared secret. Ideally, this wouldn't be required, and a tool, without constantly downloading and analyzing schemas for each and every XML document, could easily figure this out from the instance.

3. XML documents tend to be monolithic. Given a purchase order, for example, and the desire to insert a line item or replace the address, it is very hard to know how, since the items don't contain IDs or the date they were last created/updated. By contrast, the database world breaks things down into rows, each of which typically has a unique ID and a date created, although the last varies from database to database. This lets people "chunk" the information while they still have the ability to assemble it. By default in XML, the order of child elements matters. This encourages errors, however. It violates the rule of letting sloppy people author. A lot of XML errors are simple differences in the order of child elements. In most cases (except documents intended for printing) this does not and should not matter.

A HOPEFUL TURN

Recently, an opportunity has arisen to transcend these limitations. RSS 2.0 has become an extremely popular format on the Web. RSS 2.0 and Atom (which is essentially isomorphic) both support a base schema that provides a model for sets. Atom's general model is a container (a `<feed>`) of `<entry>` elements in which each `<entry>` may contain any namespace scoped elements it chooses (thus any XML), must contain a small number of required elements (`<id>`, `<updated>`, and `<title>`), and may contain some other well-known ones in the Atom namespace such as `<link>`s.

Even better, Atom clearly says that the order doesn't matter. This immediately gives a simple model for sets missing in XML. All one has to do is create a `<feed>` for a set and put each item in the set into an `<entry>`. Since all `<entry>` elements contain an `<id>` (which is a GUID) and an `<updated>` element (which is the date it was last altered), it is easy to define the semantics of replacing specific entries and even confirm that you are replacing

the ones that you think (e.g., they have the same <id> and the same <updated>. Since they have a title, it is easy to build a user interface to display menus of them. Virtually all Atom entries have a <content> or <summary> element (or both) that summarizes the value of the entry in a user-friendly way and enhances the automatic user interface. If the entry has related binary information such as pictures or sound, it contains <link> elements that have attributes describing the MIME type and size of the target, thus letting the consumer of an Atom feed make intelligent choices about which media to fetch when and how, and resolving the outage XML has in this regard.

Atom also supports links of other sorts, such as comments, so clearly an Atom entry can contain links to related feeds (e.g., Reviews for a Restaurant or Complaints for a Customer) or links to specific posts. This gives us the network and graph model that is missing in XML. Atom contains a simple HTTP-based way to INSERT, DELETE, and REPLACE <entry>s within a <feed>. There is a killer app for all these documents because the browsers already can view RSS 2.0 and Atom and, hopefully, will soon natively support the Atom protocol as well, which would mean read and write capabilities.

```
atomEntry =
  element atom:entry {
    atomCommonAttributes,
    (atomAuthor*
    & atomCategory*
    & atomContent?
    & atomContributor*
    & atomId
    & atomLink*
    & atomPublished?
    & atomRights?
    & atomSource?
    & atomSummary?
    & atomTitle
    & atomUpdated
    & extensionElement*)
  }
```

RSS 2.0 has reached the tipping point where it is universally understood. Missing from RSS is the time-to-live (TTL) indicator necessary for caching, but as discussed, HTTP provides a coarse-grained model for this. In short, it may well be that a lot of useful information is going to flow over the Web as either RSS 2.0 or Atom (or both, depending on the type the requester asks for). It addresses many of the serious limitations or outages in XML today.

Today **databases violate essentially every lesson** we have learned from the Web.

WHAT DOES THIS MEAN FOR DATABASES?

All of this has profound implications for databases. Today databases violate essentially every lesson we have learned from the Web.

1. Are simple relaxed text formats and protocols supported? No. We're still in the CORBA world. One still requires custom code to read a database. It is as though the browser required a driver to talk to each site (or at least type of site). It makes no sense in this day and age, and, as we have just seen, there is now a worthy candidate in Atom for a default protocol that every database should support, and in both Atom and RSS 2.0 for a wire format.

2. Have databases enabled people to harness Moore's law in parallel? This would mean that databases could scale more or less linearly to handle both the volume of the requests coming in and even the complexity. The answer is no. Things like ORDER BY, joins, subqueries, and many others make it almost impossible to push the query logic down to an arbitrary number of leaf nodes and simply sort/merge/aggregate the results. The easier way to limit queries to avoid this would be to limit all predicates to ones that can be computed against a single row at a time, at least where efficiency and scale are paramount.

3. Do databases optimize caching when it is OK to be stale? No. Databases in general don't have a way to mark what the TTL of fields is and then typically don't

Learning from THE WEB

have a way in the query language to describe the acceptable staleness. Thus, caching across a sea of machines, the other best mechanism for speed, is rendered almost impossible.

4. Do databases let schemas evolve for a set of items using a bottom-up consensus/tipping point? Obviously not. They typically are extremely rigid about the schema. This is regarded as a feature. Do databases let users “tag” data? Not at all easily because the point of tag rows would be that the foreign key in each row might point at *any* other row in any table, and relational databases don’t typically enable this.

5. Do databases handle flexible graphs (or trees) well? No, they do not. It has long been known that walking an arbitrary tree is a nightmare for databases. This is a big problem even today for many sites. While you can model any link that is normative in a database, it is almost impossible to model specific links to specific sets. In other words, I can have a WhoToGoTo field be a foreign key into People, but not into People or Companies or Friends or Helpdesks, or sometimes be a single value and sometimes a set and sometimes a pointer to an HTML page. Today’s databases are to graphs as IMPROV was to a spreadsheet—namely, much more formal, rigorous, and inflexible. The spreadsheet turned out to be the more robust and useful tool.

6. Have the databases learned from the Web and made their queries simple and flexible? No, just ask a database if it has anyone who, if they have an age, are older than 40; and if they have a city, live in New York; and if they have an income, earn more than \$100,000. This is a nightmare because of all the tests for NULL. There isn’t a simple effective standard for the most basic query of all, which is “about” the following words (e.g., tell me about everything: friends, people, employees, projects...). Yet this is the type of query that is most forgiving, most flexible, and most widely understood.

This is changing. Oracle has done a remarkable job of adding XML to its database in the various ways that customers might want. In so doing, it has added a lot of these capabilities. Its ROWID type allows some forms of

flexible linkage. But none really shows that they have learned from the Web.

TIME FOR DATABASE VENDORS TO STEP UP TO THE PLATE Distributed computing has been learning and evolving in response to the lessons of the Web. Formats and protocols are arising to overcome the limitations of XML—even as XML in turn arose to overcome the limitations of CORBA and DCOM. It is time that the database vendors stepped up to the plate and started to support a native RSS 2.0/Atom protocol and wire format; a simple way to ask very general queries; a way to model data that encompasses trees and arbitrary graphs in ways that humans think about them; far more fluid schemas that don’t require complex joins to model variations on a theme about anything from products to people to places; and built-in linear scaling so that the database salespeople can tell their customers, in good conscience, for this class of queries you can scale arbitrarily with regard to throughput and extremely well even with regard to latency, as long as you limit yourself to the following types of queries. Then we will know that the database vendors have joined the 21st century. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

ADAM BOSWORTH joined Google in 2004 as vice president of engineering. He came to Google from BEA Systems where he was chief architect and senior vice president of advanced development, responsible for the engineering efforts for BEA’s Framework Division. Prior to joining BEA, he co-founded Crossgain, a software development firm acquired by BEA. Known as one of the pioneers of XML, Bosworth held various senior management positions at Microsoft, including general manager of the WebData group, a team focused on defining and driving XML strategy. While at Microsoft, he was responsible for designing the Microsoft Access PC database product and assembling and driving the team that developed Internet Explorer 4.0’s HTML engine.

© 2005 ACM 1542-7730/05/1000 \$5.00