Can Web Services Scale Up?

Ken Birman, Cornell University

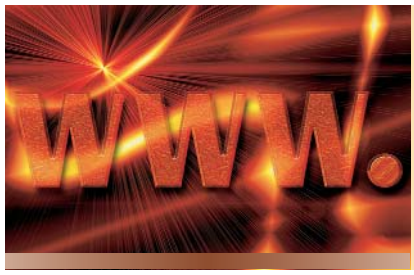
In the past, only major Internet players such as Amazon, eBay, and Google were interested in deploying large-scale Web services. However, this is changing rapidly—all sorts of companies and governmental organizations are suddenly looking towards Web services as a platform that might support a wide range of demanding applications.

Examples of such systems include big banking and brokerage data centers, online service centers for companies that operate on a global scale, systems that operate critical infrastructures like electric power and transportation, and government and military systems responsible for everything from intelligence gathering to issuing Social Security checks.

This emerging trend presents developers with a new challenge: building Web services solutions that scale.

In a nutshell, a scalable system is one that can flexibly accommodate growth in its client base. Such systems typically run on a clustered computer or in a large data center and must be able to handle high loads or sudden demand bursts and a vast number of users. They must reliably respond even in the event of failures or reconfiguration. Ideally, they're self-managed and automate as many routine services such as backups and component upgrades as possible. Many settings also require security against attempted intrusions and distributed denial-of-service (DDoS) attacks.

At a glance, today's Web services standards seem to answer these needs. However, a more probing analysis reveals many critical limitations.



Current Web services standards have many critical limitations.

CURRENT LIMITATIONS

Consider, for example, the major Web services standards dealing with reliability: WS-Reliability and WS-Transactions.

WS-Reliability provides for reliable handoff between a client system and a queuing system residing between the client and some service. However, the standard isn't nearly as comprehensive as the name implies; rather, it's limited to pipelines that include queuing subsystems. WS-Reliability boils down to a few options that a client can use to tell the queuing system whether or not to reissue a request if a failure occurs, and a way to timestamp requests so that a service can detect duplicates.

WS-Transactions actually consists of two side-by-side standards: One is aimed at applications that perform database transactions with the usual ACID (atomicity, consistency, isolation, durability) properties; the second builds on the first and supports a way to build "scripts" of simpler transactions.

Some might argue that all reliability needs can be recast in terms of transactions. However, the past three decades have seen one failed attempt after another to build everything over a database system, and it's now clear that many kinds of systems just don't match the model.

These intrinsically distributed systems make use of direct communication between programs—via TCP, publish-subscribe protocols, or RPC—that can't tolerate delay. These systems lack databases' clean separation of stored data from code, and any attempt to force them into that model results in unacceptable loss of performance.

Intrinsically distributed systems are common, and Web services will need to support them. However, the existing reliability options simply don't address the requirement.

A LESSON FROM THE PAST

What sorts of scaling and reliability features are lacking in Web services standards today? A good example is data replication: Building a server that scales to handle load often requires replicating data on multiple nodes of a cluster. Given replication, another example is guaranteed real-time responsiveness: A company that buys a cluster probably wants to guarantee

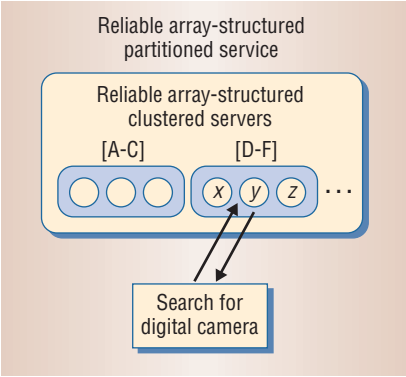


Figure 1. Example of RAPS of RACS. The service assigns a digital camera search request to the clustered server handling all Ds, and a load balancer routes it to the appropriate process.

that some service will be responsive enough to keep its customers happy even when demand is high.

The missing technologies don't stop there. What about life-cycle services that can launch an application on demand or restart a failed component? Or load balancers and technology to automate management of a machine cluster running Web services applications?

There are some 50 working groups within the World Wide Web Consortium (www.w3.org), the primary organization developing Web services standards. However, not one is addressing these kinds of issues.

A similar dynamic played out in the early 1980s, when client-server computing was touted as the next big thing, a silver bullet to solve every problem related to older mainframe and batch systems. Companies rushed to move everything from mainframe settings to client-server configurations. There were notable successes, but it quickly became apparent that the early platforms were strikingly immature.

Processes needed to be automated and standardized, and the early generations of client-server systems cost a fortune to build, were unreliable, required armies of systems administrators and specialists, and were extremely insecure. The "total cost of ownership" proved to be unexpectedly and unacceptably high.

The lesson of the client-server era is that incomplete platforms can't support major, complex applications. My concern is that the Web services community is about to face the same problem. Platform developers are racing forward at top speed, jostling for position with ever more exaggerated claims, while closing their eyes to the dangerous potholes in the road ahead.

ARCHITECTURAL STANDARDS FOR SCALABILITY

To properly address scalability in Web services, we need more than a long list of reliability and management standards; we need a new methodology suitable for supporting a scalable data center architecture.

Jim Gray, the 1994 Turing Award winner, along with Pat Helland and Dennis Shasha, recommends that developers think in terms of a reliable array-structured partitioned service (RAPS) implemented as a set of reliable array-structured clustered servers (RACS).

This architecture offers scalability and reliability at two levels. The top level uses some sort of application-specific key to partition the service into subservices. The lower level implements subservices using groups of programs that run on multiple machines, perhaps in a cluster computer. The groups replicate data so that each can handle any incoming query for its range within the keys, enabling updates to reach all the replicas.

Consider, for example, a RAPS that an e-tailer such as Amazon might use to personalize a product recommendation. Depending on the customer's profile, the service ranks matching products differently to maximize the chance of a purchase. If the product is, say, a digital camera, as shown in Figure 1, the service assigns the search request to the RACS handling all Ds (other keys, such as the customer's name, are equally plausible). The load balancer then routes the request to the appropriate program for processing—in this case, process *y*.

With support for this basic layout, it's possible to tackle a wide range of secondary issues. For example, we could create standards for a self-managed RAPS of RACS, or for one that guarantees real-time response. Such a basic architecture is effectively a framework to resolve other related issues.

PROCESS-GROUP REPLICATION

Web services currently lacks support for building scalable services. The architecture makes it easy to build a single-node server that responds to requests from some set of clients, but there's no way to turn that single server into a RACS or turn a set of RACS into a RAPS. However, it would be easy to bridge the gap if vendors and platform builders wanted to do so.

Old and familiar technologies

The most standard form of system support for building a RAPS of RACS would draw on *virtual synchrony,* a process-group computing model developed at Cornell in the 1980s and used today to run the New York and Swiss stock exchange systems, the French air traffic control system, and the US Navy's Aegis-class warship. IBM's Websphere platform and the Windows Vista clustering system also use versions of the model, although developers can't access the internal mechanisms directly.

The other popular standard uses a *state-machine* approach to guarantee stronger durability. Leslie Lamport's Paxos algorithm, which is implemented in scalable file systems and other ultrareliable server designs, exemplifies this approach.

One architecture could support both of these powerful technologies. A natural option would be to offer them in the context of WS-Eventing, the publish-subscribe standard. After all, if you're replicating data within some form of "group," you can just as easily imagine that it has a "subject name" in a publish-subscribe context.

Advantages

With this type of process-group formation, data can be anything: files, other kinds of persistent objects, or even in-memory data structures. The user simply designs a data structure and employs multicast technology to transmit updates to the group members, which apply them in the same order everywhere (a read, in contrast, can be done on any desired copy).

Examples of updates include a stock trade or stock market quote, a new object detected by radar in an air traffic control system, a communication to or from an aircraft, or the addition of a node to a distributed data structure containing an index of pending orders in an online warehouse.

Data replication can be remarkably cheap: With modern technology and small updates, a four-node virtual synchrony service can run at rates well in excess of 200,000 asynchronous (yet fault-tolerant) ordered updates per second. Even if an update requires a large message, it's possible to maintain rates of thousands per second on typical hardware.

The virtual synchrony and state-machine models show how a tremendous range of application requirements can map down to a rigorously precise execution model, which in turn can be used to validate a platform. Because the models have formal specifications, you can test the correctness of an implementation, and even use theorem provers to assist developers in testing their most critical application components.

Concerns

One reason that we lack this sort of support today is that vendors and platform developers worry that these forms of replication haven't achieved huge market success.

As the "Experience with Corba" sidebar describes, the common object request broker architecture offers a fault-tolerant groups mechanism that was based on the virtual synchrony model. Unfortunately, the Corba standard is widely viewed as rigid and limited. I believe that the Corba community erred by embedding a powerful solution into a tool mismatched to developer needs.

The Corba community's failed effort to implement virtual synchrony carries an important lesson to current researchers: Any technology offered to developers must support the programming styles they prefer.

Experience with Corba

Even good ideas can be used in ways that developers dislike and ultimately reject. A good example of this occurred when the Corba community decided to tackle replication for fault tolerance but then stumbled by presenting the technology to developers in a way that was much too limiting for general use.

Corba's fault-tolerance mechanism is based on the virtual synchrony model, but the programming tools built "over" this model prevent developers from using threads, certain I/O operations, GUIs or other direct end-user interactions, shared memory, debugging tools, logging, interrupts, the clock, or even prebuilt libraries.

In the Corba approach, a developer who obeys this long list of constraints can do lockstep replication of a program for tolerance of hardware faults. Unfortunately, though, the scheme doesn't protect against software-related crashes.

Not surprisingly, developers regard the standard as rigid and limited. They need fault tolerance, but not in this very narrow form.

In contrast, systems like the Isis toolkit, popular during the early and mid-1990s, also used virtual synchrony but had fewer limitations. They supported many of the mechanisms needed to build and manage a RAPS of RACS, and their successes have clearly demonstrated the model's effectiveness.

Isis is no longer available as a product, yet many critical systems continue to use Isis-based solutions or other virtual synchrony implementations. The state-machine approach as used in the Paxos algorithm is also becoming more popular.

The key insight is that these successes use similar ideas but in ways very different from what the Corba fault-tolerance standard requires. What we need today is a modern revisiting of this technology that draws on group communication but packages it in a way that developers perceive as solving their most pressing scalability problems and that flexibly matches their preferred styles and tools.

MANAGEMENT POLICIES

A scalable services architecture for building RAPS of RACS alone isn't enough. Large-scale systems that will likely soon rely on standardized Web services—including global banks, the entire US Air Force, and the supervisory control and data acquisition systems that operate the US power grid—will also require policies to manage security keys, firewalls, service life cycles, and so on.

Automated tools for monitoring large complex systems will be needed as well. Finally, researchers must think about how monitoring and management policies in different organizations should "talk" to one another when Web services interactions cross boundaries.

These are tough problems, but they can be solved. For example, at Cornell we recently developed Astrolabe, a scalable technology for distributed monitoring and control that has attracted tremendous interest and attention.

Researchers at other institutions are working on other promising solutions.

Scalability isn't just a technology; it's also a mindset with ramifications at many levels. To ensure true scalability, Web services platforms must begin to standardize application architectures that promote reliability and interoperability when developers build "systems of systems," work with intrinsically distributed programs that don't fit a transactional model, and must provide responsiveness guarantees to their users.

Applications with these sorts of requirements are already in the pipeline and even more of them are on drawing boards in government, corporate, and military settings. The only option for the Web services community is to take on the challenge; if they do so, solutions will be readily available.

Web services are going to be the ubiquitous platform technology for next-generation critical computing systems, and we've no one but ourselves to blame if these systems don't work properly. Do we really want to create a world in which minor computer glitches shut down massive critical applications and in which hackers can readily disrupt access to banking records, air traffic control systems, and even shut down the power grid?

Time is running out. Current halfway solutions will tempt developers to embark on a path that will soon lead many of them into real trouble. The entire industry—clients, developers, and vendors—as well as the government have a shared obligation to make Web services better. ■

Ken Birman *is a professor in the Department of Computer Science at Cornell University. Contact him at ken@cs.cornell.edu.*

Editor: Simon S.Y. Shim, Department of Computer Engineering, San Jose State University; sishim@email.sjsu.edu