Amazon Architecture

Tuesday, September 18, 2007 at 5:44AM

Todd Hoff in Example, Java, Linux, Oracle, Perl

This is a wonderfully informative Amazon update based on Joachim Rohde's discovery of an interview with Amazon's CTO. You'll learn about how Amazon organizes their teams around services, the CAP theorem of building scalable systems, how they deploy software, and a lot more. Many new additions from the ACM Queue article have also been included.

Amazon grew from a tiny online bookstore to one of the largest stores on earth. They did it while pioneering new and interesting ways to rate, review, and recommend products. Greg Linden shared is version of Amazon's birth pangs in a series of blog articles

Site: http://amazon.com

Information Sources

- Early Amazon by Greg Linden
- How Linux saved Amazon millions
- Interview Werner Vogels - Amazon's CTO
- Asynchronous Architectures - a nice summary of Werner Vogels' talk by Chris Loosley
- Learning from the Amazon technology platform - A Conversation with Werner Vogels
- Werner Vogels' Weblog - building scalable and robust distributed systems

Platform

- Linux
- Oracle
- C++
- Perl
- Mason
- Java
- Jboss
- Servlets

The Stats

- More than 55 million active customer accounts.
- More than 1 million active retail partners worldwide.
- Between 100-150 services are accessed to build a page.

The Architecture

- What is it that we really mean by scalability? A service is said to be scalable if when we increase the resources in a system, it results in increased performance in a manner proportional to resources added. Increasing performance in general means serving more units of work, but it can also be to handle larger units of work, such as when datasets grow.

- The big architectural change that Amazon made was to move from a two-tier monolith to a fully-distributed, decentralized, services platform serving many different applications.
- Started as one application talking to a back end. Written in C++.
- It grew. For years the scaling efforts at Amazon focused on making the back-end databases scale to hold more items, more customers, more orders, and to support multiple international sites. In 2001 it became clear that the front-end application couldn't scale anymore. The databases were split into small parts and around each part and created a services interface that was the only way to access the data.

- The databases became a shared resource that made it hard to scale-out the overall business. The front-end and back-end processes were restricted in their evolution because they were shared by many different teams and processes.
- Their architecture is loosely coupled and built around services. A service-oriented architecture gave them the isolation that would allow building many software components rapidly and independently.
- Grew into hundreds of services and a number of application servers that aggregate the information from the services. The application that renders the Amazon.com Web pages is one such application server. So are the applications that serve the Web-services interface, the customer service application, and the seller interface.
- Many third party technologies are hard to scale to Amazon size. Especially communication infrastructure technologies. They work well up to a certain scale and then fail. So they are forced to build their own.
- Not stuck with one particular approach. Some places they use jboss/ java, but they use only servlets, not the rest of the J2EE stack.
- C++ is uses to process requests. Perl/Mason is used to build content.
- Amazon doesn't like middleware because it tends to be framework and not a tool. If you use a middleware package you get lock-in around the software patterns they have chosen. You'll only be able to use their software. So if you want to use different packages you won't be able to. You're stuck. One event loop for messaging, data persistence, AJAX, etc. Too complex. If middleware was available in smaller components, more as a tool than a framework, they would be more interested.
- The SOAP web stack seems to want to solve all the same distributed systems problems all over again.
- Offer both SOAP and REST web services. 30% use SOAP. These tend to be Java and .NET users and use WSDL files to generate remote object interfaces. 70% use REST. These tend to be PHP or PERL users.
- In either SOAP or REST developers can get an object interface to

Amazon. Developers just want to get job done. They don't care what goes over the wire.

- Amazon wanted to build an open community around their services. Web services were chosed because it's simple. But hat's only on the perimeter. Internally it's a service oriented architecture. You can only access the data via the interface. It's described in WSDL, but they use their own encapsulation and transport mechanisms.
- Teams are Small and are Organized Around Services

- Services are the independent units delivering functionality within Amazon. It's also how Amazon is organized internally in terms of teams.
- If you have a new business idea or problem you want to solve you form a team. Limit the team to 8-10 people because communication hard. They are called two pizza teams. The number of people you can feed off two pizzas.
- Teams are small. They are assigned authority and empowered to solve a problem as a service in anyway they see fit.
- As an example, they created a team to find phrases within a book that are unique to the text. This team built a separate service interface for that feature and they had authority to do what they needed.
- Extensive A/B testing is used to integrate a new service . They see what the impact is and take extensive measurements.

- Deployment

- They create special infrastructure for managing dependencies and doing a deployment.
- Goal is to have all right services to be deployed on a box. All application code, monitoring, licensing, etc should be on a box.
- Everyone has a home grown system to solve these problems.
- Output of deployment process is a virtual machine. You can use EC2 to run them.

- Work From the Customer Backwards to Verify a New Service is Worth Doing

- Work from the customer backward. Focus on value you want to deliver

for the customer.
- Force developers to focus on value delivered to the customer instead of building technology first and then figuring how to use it.
- Start with a press release of what features the user will see and work backwards to check that you are building something valuable.
- End up with a design that is as minimal as possible. Simplicity is the key if you really want to build large distributed systems.

- State Management is the Core Problem for Large Scale Systems

- Internally they can deliver infinite storage.
- Not all that many operations are stateful. Checkout steps are stateful.
- Most recent clicked web page service has recommendations based on session IDs.
- They keep track of everything anyway so it's not a matter of keeping state. There's little separate state that needs to be kept for a session. The services will already be keeping the information so you just use the services.

- Eric Brewer's CAP Theorem or the Three properties of Systems

- Three properties of a system: consistency, availability, tolerance to network partitions.
- You can have at most two of these three properties for any shared-data system.
- Partitionability: divide nodes into small groups that can see other groups, but they can't see everyone.
- Consistency: write a value and then you read the value you get the same value back. In a partitioned system there are windows where that's not true.
- Availability: may not always be able to write or read. The system will say you can't write because it wants to keep the system consistent.
- To scale you have to partition, so you are left with choosing either high consistency or high availability for a particular system. You must find the right overlap of availability and consistency.
- Choose a specific approach based on the needs of the service.

- For the checkout process you always want to honor requests to add items to a shopping cart because it's revenue producing. In this case you choose high availability. Errors are hidden from the customer and sorted out later.
- When a customer submits an order you favor consistency because several services--credit card processing, shipping and handling, reporting--are simultaneously accessing the data.

Lessons Learned

- You must change your mentality to build really scalable systems. Approach chaos in a probabilistic sense that things will work well. In traditional systems we present a perfect world where nothing goes down and then we build complex algorithms (agreement technologies) on this perfect world. Instead, take it for granted stuff fails, that's reality, embrace it. For example, go more with a fast reboot and fast recover approach. With a decent spread of data and services you might get close to 100%. Create self-healing, self-organizing lights out operations.

- Create a shared nothing infrastructure. Infrastructure can become a shared resource for development and deployment with the same downsides as shared resources in your logic and data tiers. It can cause locking and blocking and dead lock. A service oriented architecture allows the creation of a parallel and isolated development process that scales feature development to match your growth.

- Open up you system with APIs and you'll create an ecosystem around your application.

- Only way to manage as large distributed system is to keep things as simple as possible. Keep things simple by making sure there are no hidden requirements and hidden dependencies in the design. Cut technology to the minimum you need to solve the problem you have. It doesn't help the company to create artificial and unneeded layers of complexity.

- Organizing around services gives agility. You can do things in parallel is because the output is a service. This allows fast time to market. Create an infrastructure that allows services to be built very fast.

- There's bound to be problems with anything that produces hype before real implementation

- Use SLAs internally to manage services.

- Anyone can very quickly add web services to their product. Just implement one part of your product as a service and start using it.

- Build your own infrastructure for performance, reliability, and cost control reasons. By building it yourself you never have to say you went down because it was company X's fault. Your software may not be more reliable than others, but you can fix, debug, and deployment much quicker than when working with a 3rd party.

- Use measurement and objective debate to separate the good from the bad. I've been to several presentations by ex-Amazoners and this is the aspect of Amazon that strikes me as uniquely different and interesting from other companies. Their deep seated ethic is to expose real customers to a choice and see which one works best and to make decisions based on those tests.

Avinash Kaushik calls this getting rid of the influence of the HiPPO's, the highest paid people in the room. This is done with techniques like A/B testing and Web Analytics. If you have a question about what you should do code it up, let people use it, and see which alternative gives you the results you want.

- Create a frugal culture. Amazon used doors for desks, for example.

- Know what you need. Amazon has a bad experience with an early recommender system that didn't work out: "This wasn't what Amazon needed. Book recommendations at Amazon needed to work from sparse data, just a few ratings or purchases. It needed to be fast. The system needed to scale to massive numbers of customers and a huge catalog. And it needed to enhance discovery, surfacing books from deep in the catalog that readers wouldn't find on their own."

- People's side projects, the one's they follow because they are interested, are often ones where you get the most value and innovation. Never underestimate the power of wandering where you are most interested.

- Involve everyone in making dog food. Go out into the warehouse and pack books during the Christmas rush. That's teamwork.

- Create a staging site where you can run thorough tests before releasing into the wild.

- A robust, clustered, replicated, distributed file system is perfect for read-only data used by the web servers.

- Have a way to rollback if an update doesn't work. Write the tools if necessary.

- Switch to a deep services-based architecture (http://webservices.sys-con.com/read/262024.htm).

- Look for three things in interviews: enthusiasm, creativity, competence. The single biggest predictor of success at Amazon.com was enthusiasm.

- Hire a Bob. Someone who knows their stuff, has incredible debugging skills and system knowledge, and most importantly, has the stones to tackle the worst high pressure problems imaginable by just leaping in.

- Innovation can only come from the bottom. Those closest to the problem are in the best position to solve it. any organization that depends on innovation must embrace chaos. Loyalty and obedience are not your tools.

- Creativity must flow from everywhere.

- Everyone must be able to experiment, learn, and iterate. Position, obedience, and tradition should hold no power. For innovation to flourish, measurement must rule.

- Embrace innovation. In front of the whole company, Jeff Bezos would give an old Nike shoe as "Just do it" award to those who innovated.

- Don't pay for performance. Give good perks and high pay, but keep it flat. Recognize exceptional work in other ways. Merit pay sounds good but is almost impossible to do fairly in large organizations. Use non-monetary awards, like an old shoe. It's a way of saying thank you, somebody cared.

- Get big fast. The big guys like Barnes and Nobel are on your tail. Amazon wasn't even the first, second, or even third book store on the web, but their vision and drive won out in the end.

- In the data center, only 30 percent of the staff time spent on infrastructure issues related to value creation, with the remaining 70 percent devoted to dealing with the "heavy lifting" of hardware procurement, software management, load balancing, maintenance, scalability challenges and so on.

- Prohibit direct database access by clients. This means you can make you service scale and be more reliable without involving your clients.

This is much like Google's ability to independently distribute improvements in their stack to the benefit of all applications.

- Create a single unified service-access mechanism. This allows for the easy aggregation of services, decentralized request routing, distributed request tracking, and other advanced infrastructure techniques.

- Making Amazon.com available through a Web services interface to any developer in the world free of charge has also been a major success because it has driven so much innovation that they couldn't have thought of or built on their own.

- Developers themselves know best which tools make them most productive and which tools are right for the job.

- Don't impose too many constraints on engineers. Provide incentives for some things, such as integration with the monitoring system and other infrastructure tools. But for the rest, allow teams to function as independently as possible.

- Developers are like artists; they produce their best work if they have the freedom to do so, but they need good tools. Have many support tools that are of a self-help nature. Support an environment around the service development that never gets in the way of the development itself.

- You build it, you run it. This brings developers into contact with the day-to-day operation of their software. It also brings them into day-to-day contact with the customer. This customer feedback loop is essential for improving the quality of the service.

- Developers should spend some time with customer service every two years. Their they'll actually listen to customer service calls, answer customer service e-mails, and really understand the impact of the kinds of

things they do as technologists.

- Use a "voice of the customer," which is a realistic story from a customer about some specific part of your site's experience. This helps managers and engineers connect with the fact that we build these technologies for real people. Customer service statistics are an early indicator if you are doing something wrong, or what the real pain points are for your customers.

- Infrastructure for Amazon, like for Google, is a huge competitive advantage. They can build very complex applications out of primitive services that are by themselves relatively simple. They can scale their operation independently, maintain unparalleled system availability, and introduce new services quickly without the need for massive reconfiguration.
