The Canonical Cloud Architecture

Friday, August 7, 2009 at 12:06AM

Todd Hoff in Example, cloud

Update 2: Elastic Load Balancer and EC2 instance bandwidth. *It turns out we are limited by bandwidth and not by CPU*. Solution: *use DNS Round Robin for two to three HighCPU medium instances*.
Update: The Skinny Straw: Cloud Computing's Bottleneck and How to Address It. *For cloud computing, bandwidth to and from the cloud provider is a bottleneck*. Solution: *Evaluate application architecture and consider application partitioning*.

I'm writing this post as a sort of penance. My sin was getting involved in another mutli-threaded mess of a program that was rife with strange pauses and unexpected errors. I really should have known better. But when APIs choose to make callbacks from some mystery thread pool it's hard to keep things straight. I eventually sobered up and posted all events to a queue so I could make sure the program would work correctly. Doh. I may never know why the .Net console output stopped working, but I'll live with it.

And that reminded me that I've been meaning to write a post on the standard Cloud Architecture. I've tried to hit all the common architectures at one time or another, but there have been some excellent sources lately on structuring programs in a cloud that people may "know" in the same way I knew what not to do, but when the code hits the editor those thoughts may have hidden like a kid next to a broken cookie jar.

The easiest way to create a scalable service is to compose the service from other scalable services. This is how Google AppEngine works and is largely how AWS works as well (EC2, S3, SQS, SimpleDB, etc), though AWS also functions as a blank canvas on which you can draw your own designs.

The canonical cloud architecture that has evolved revolves around dynamically scalable CPUs consuming asynchronous, persistently queued events. We talked about this idea already in Flickr - Do the Essential Work Up-front and Queue the Rest. The cloud is just another way of implementing the same idea.

Amazon suggests a few applications of the Cloud Architecture as:

- Processing Pipelines

- Document processing pipelines – convert hundreds of thousands of documents from Microsoft Word to PDF, OCR millions of pages/images into raw searchable text
- Image processing pipelines – create thumbnails or low resolution variants of an image, resize millions of images
- Video transcoding pipelines – transcode AVI to MPEG movies
- Indexing – create an index of web crawl data
- Data mining – perform search over millions of records

- Batch Processing Systems

- Back-office applications (in financial, insurance or retail sectors)
- Log analysis – analyze and generate daily/weekly reports
- Nightly builds – perform nightly automated builds of source code repository every night in parallel
- Automated Unit Testing and Deployment Testing – Test and deploy and perform automated unit testing (functional, load, quality) on different deployment configurations every night

- Websites

- Websites that "sleep" at night and auto-scale during the day
- Instant Websites – websites for conferences or events (Super Bowl, sports tournaments)
- Promotion websites
- "Seasonal Websites" - websites that only run during the tax season or the holiday season ("Black Friday" or Christmas)

A good list, but after having worked on a seasonal website for taxes AWS is a horrible match. AWS only works on the instance level, so you need a whole instance turned on all the time even when there's no demand. This is a complete waste of money. An AWS model truly based on use combined with an SLA driven dashboard would be very convenient. But on to cases where AWS is a good fit.

SmugMug's Cloud Architecture

AWS pioneer Don MacAskill of SmugMug details how they process high-resolution photos and high-definition video use a cloud hosted queuing architecture in SkyNet Lives! (aka EC2 @ SmugMug).

SkyNet, as you might expect, operates completely without human minders and automatically scales up and down in relation to the work load. Their system has several components:

- **Work Initiators -** Work comes in from your website and/or other software subsystems and is queued up for processing in the Queue Service. Work doesn't have to be large requests either. Work can be small independent parts of an overall pipeline. Don't keep state in the Workers. Bundle what you need done into a work request in shoot back into the Queuing Service for processing.
- **Provisioning Service -** This is Amazon's infrastructure that allows instances to be automatically scaled up and down in relation to the work load. This will be the major difference between your VPS or typical datacenter setup. There's an API for starting and stopping AMIs and
mechanisms for automatically configuring and running VMs.
- **Workers -** These are the guys that continually pull work off queues and do something interesting with it. For SmugMug the results are stored on S3 but the results could be put in your own database, SimpleDB or whatever.
- **Queuing Service -** This is where work is queued for consumption by the workers. SmugMug built their own queuing service, but you could just as easily use Amazon's own SQS. Creating a scalable, distributed, performant, highly available queue service is not easy, so you may want to take a look at a number of different queue product suggestions in Flickr - Do the Essential Work Up-front and Queue the Rest.
- **Controller -** This component monitors many variables related to the work flow and decides how many instances of EC2 are necessary based on optimizing a small set of goals. Instances are add and removed as needed.

Don shares a lot of practical detailia on how to efficiently use AWS, how their queue service works, and how their controller manages to balances minimizing cost while still being responsive to users. Achieving fairness and balance in a queue system can be difficult, but SmugMug appears to have done a good job of that.

What rocks about queuing architectures is that they are just so damn robust. Work is safe in the queues. A random reboot won't cause a loss. If one component is producing events too fast the queue will buffer up events until they can be processed. New components can be cleanly added and removed from the system at any time. Timing isn't critical. Work is processed when someone gets around to it. Timeouts and retries are unnecessary. Programs are simple loops that block on the queue, do something, persist results, and feed back more parallelizable work requests back into the queue. Very hard to screw up. Compare and contrast to complex multi-threaded system with shared-state.

Building GrepTheWeb in the Cloud

Amazon has published a great couple of articles on building a canonical Cloud Architecture: Building GrepTheWeb in the Cloud, Part 1: Cloud Architectures and Building GrepTheWeb in the Cloud, Part 2: Best Practices.

These are really tight and well written articles so I'll just hit certain high points. The example used is an application called GrepTheWeb. GrepTheWeb searches using a regular expression across millions of web documents. So it's a grep for the web, ah got it now. The idea is to take an unpredictable but possibly large number of search requests, apply the search expression to hundreds of terabytes of documents, and return the results in a reasonable period of time.

How exactly would you do such a thing? Here's how you do it in the cloud:
- Amazon S3 for retrieving input datasets and for storing the output dataset
- Amazon SQS for durably buffering requests acting as a “glue” between controllers
- Amazon SimpleDB for storing intermediate status, log, and for user data about tasks
- Amazon EC2 for running a large distributed processing Hadoop cluster on-demand
- Hadoop for distributed processing, automatic parallelization, and job scheduling

Clearly these are all (except for Hadoop) built on Amazon services, but the general ideas apply anywhere. For storing large amounts of data and accessing it efficiently in parallel you need a distributed file system like S3. To coordinate and dispatch work you need a queuing service like SQS. For keeping intermediate state you need a scalable database store like SimpleDB, though you could also imagine using S3. For dynamically scaling processing nodes something like EC2 is necessary. And for actually carrying out the document search a framework like Hadoop provides a lot of features, though you can imagine using other compute grid products.

Here's their fabulous picture of what the system looks like:

All the parts and linkages are described in the paper. What's important to note is that even though there are a lot of independently moving parts all the boundaries are clear and well described. In your typical program few will have any idea how it works. Using Cloud Architecture principles it's possible to create a system which both scales and easy to understand and explain.

The paper makes several key architectural recommendations:

- **Use Scalable Ingredients -** Ensure that your application is scalable by designing each component to be scalable on its own. If every component implements a service interface, responsible for its own scalability in all appropriate dimensions, then the overall system will have a scalable base.
- **Have Loosely Coupled Systems -** For better manageability and high-availability, make sure that your components are loosely coupled. The key is to build components without having tight dependencies between each other, so that if one component were to die (fail), sleep (not respond) or remain busy (slow to respond) for some reason, the other components in the system are built so as to continue to work as if no failure is happening.
- **Think Parallel -** Implement parallelization for better use of the infrastructure and for performance. Distributing the tasks on multiple machines, multithreading your requests and effective aggregation of results obtained in parallel are some of the techniques that help exploit the infrastructure.
- **Utilize On-Demand Requisition and Relinquishment -** After designing the basic functionality, ask the question "What if this fails?" Use techniques and approaches that will ensure resilience. If any component fails (and failures happen all the time), the system should automatically alert, failover, and re-sync back to the "last known state" as if nothing had failed.
- **Use Designs that Are Resilient to Reboot and Re-Launch -** Don't forget the cost factor. The key to building a cost-effective application is using on-demand resources in your design. It's wasteful to pay for infrastructure that is sitting idle.

All good stuff which is why I like this paper so much. There's a big conceptual shift here, especially of you are used to relatively simple client-server and N-tier systems. It's like simulating in your mind how to keep an army of ants all working independently while still communicating, coordinating, and making progress on a goal. We implemented similar architecture in datacenters long before the cloud, it was just a lot harder as everything was roll your own. The cloud makes all the necessary components standard, featureful, and relatively inexpensive. This opens any application to completley different ways of structuring their backends than they did in the past.

Related Articles

- SkyNet Lives! (aka EC2 @ SmugMug).
- Flickr - Do the Essential Work Up-front and Queue the Rest
- Hadoop
- GridGain: One Compute Grid, Many Data Grids
- Building GrepTheWeb in the Cloud, Part 1: Cloud Architectures
- Building GrepTheWeb in the Cloud, Part 2: Best Practices.
