Programming for the Amazon EC2 cloud

LOFTY CODE

We show you some techniques for harnessing the benefits of cloud technology. BY DAN FROST

Everyone is talking about the promise of cloud computing, but when it comes to implementation, some of the early adopters have simply deployed cloud services by copying older methods used in conventional environments. In fact, the cloud can do much more for you. Running sites on EC2 is easy, but really making use of the scalability and flexibility of cloud computing requires a new approach (Figure 1). In this article, I describe some techniques for building the benefits of cloud computing into your infrastructure. Although I use examples based on the Ruby language and Amazon's EC2 cloud environment, these concepts also apply to other languages and cloud vendors.

Keep It Static

In the cloud, you don't need everything to go through *your* server (even if it is virtual). You don't need a virtual server for serving files, managing queues, and storing shared data. Dedicated services can perform these tasks, and using them will help you get your applications working better in the cloud.

Online Storage

In this first example, you use an online storage service to host your static files. Because it takes an unnecessary load off of your web servers, online storage is good practice for any site operating within the cloud paradigm. In the case of the Amazon environment, the S3 service (Simple Storage Service) will play host to your static files.

Let's suppose you have a simple Ruby application, such as a blog or wiki. When your users upload a file, it is usually stored on the filesystem; instead, you could push the file to S3 directly.

To do this in Ruby, start by installing the library:

```
sudo gem install aws-s3
```

Then create a simple script like the one shown in Listing 1.

Pushing a file to S3 and making it public takes just one line:

```
AWS::S3::S3Object.store(
  'example.jpg',
  open('example.jpg'),
  'my-public-bucket',
  :access => :public_read
)
```

Of course, the URL will be different, so you must change the link in the blog post. The preceding example creates the URL:

```
http://s3.amazonaws.com/
my-public-bucket/example.jpg
```

Iakov Kalinin, Fotolia

Pushing all your static files to S3 is relatively simple – you can think of S3 as a huge static file server. More interesting is SQS, which really takes you into solving problems in a scalable way.

SQS

SQS is a queue server, which hosts a queue of data that applications can add to and remove from. This apparently trivial task makes scaling big tasks easy. Instead of needing to run all your tasks in one place and keeping everything coordinated, you can push a list of tasks onto the queue, fire up a dozen servers, and watch them work through the queue.

For example, imagine that you need to prepare a large number of personalized recommendations for customers. In a normal LAMP environment, you would need to work through a list of user records, create a set of recommendations, and store the information in a second database table. With SQS, you can split the process. In other words, you can "decouple" the process by pushing the information onto the queue in the first script and then processing the data in the queue in the second script.

Working in Rails, you can install the SQS bindings for Ruby and push a model onto the queue using the *to_xml* method:

```
q = SQS.get_queue ➘
"work-out-some-recommendations"
q.send_message myobject.to_xml
```

This code means that an XML entry in the *work-out-some-recommendations* queue will look something like the following:

```
  <myobject>
    <user>Mr. Smith</user>
    <favorite_products>
      <product>2412</product>
      <product>9374</product>
      <product>1029</product>
    </favorite_products>
  </myobject>
```

Next, you need to get this XML entry out of the queue and do something with it:

```
q = SQS.get_queue ➘
"work-out-some-recommendations"
queue_item = q.receive_message
work_object = MyObject.new()
work_object.from_xml ➘
queue_item.body
```

Work_object is just the same as *myobject* was above, but with an important difference: You don't need to connect to the original database, so you won't have any issues with the number of connections and the speed of the database server.

You're free to use the XML to build a message, which you can then push to S3 for later use by any other part of the application (Listing 2). Notice that this message isn't public. Because you are only going to use them internally, there is no need to expose these snippets.

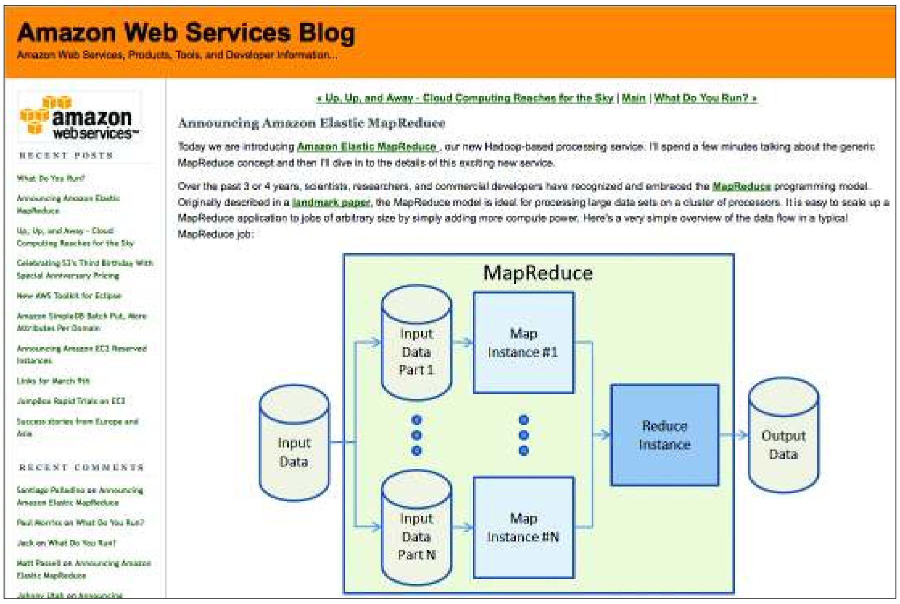
When building your web pages, you can save a few CPU cycles by pulling the welcome message from S3 rather than connecting to any other server:

```
cached_snippet = ➘
AWS::S3::S3Object.find ➘
'Welcome-dan@example.com', ➘
'welcome-messages'
```

All I have really done here is caching. Using SQS and S3 provides a completely scalable way of caching that doesn't affect the performance of your site at all.

SimpleDB – Scalable DB

One final service to consider is SimpleDB – a hugely scalable database. Amazon provides a free tier of pricing,


Amazon Web Services Blog

Amazon Web Services, Products, Tools, and Developer Information...

RECENT POSTS

What Do You Run?

Announcing Amazon Elastic MapReduce

Up, Up, and Away - Cloud Computing Reaches for the Sky

Celebrating S3's Third Birthday With Special Anniversary Pricing

New AWS Toolkit for Eclipse

Amazon SimpleDB Batch Put, More Attributes Per Domain

Announcing Amazon EC2 Reserved Instances

Links for March 9th

JumpBox Rapid Trials on EC2

Success stories from Europe and Asia

RECENT COMMENTS

Santiago Palladino on Announcing Amazon Elastic MapReduce

Paul Morriss on What Do You Run?

Jack on What Do You Run?

Matt Passell on Announcing Amazon Elastic MapReduce

Johnny Utah on Announcing

« Up, Up, and Away - Cloud Computing Reaches for the Sky | Main | What Do You Run? »

Announcing Amazon Elastic MapReduce

Today we are introducing Amazon Elastic MapReduce , our new Hadoop-based processing service. I'll spend a few minutes talking about the generic MapReduce concept and then I'll dive in to the details of this exciting new service.

Over the past 3 or 4 years, scientists, researchers, and commercial developers have recognized and embraced the MapReduce programming model. Originally described in a landmark paper, the MapReduce model is ideal for processing large data sets on a cluster of processors. It is easy to scale up a MapReduce application to jobs of arbitrary size by simply adding more compute power. Here's a very simple overview of the data flow in a typical MapReduce job:

Figure 1: Amazon recently added MapReduce to their list of services. As if cache, computing, and queues aren't enough, now you can run huge, distributed tasks.

Interoperability

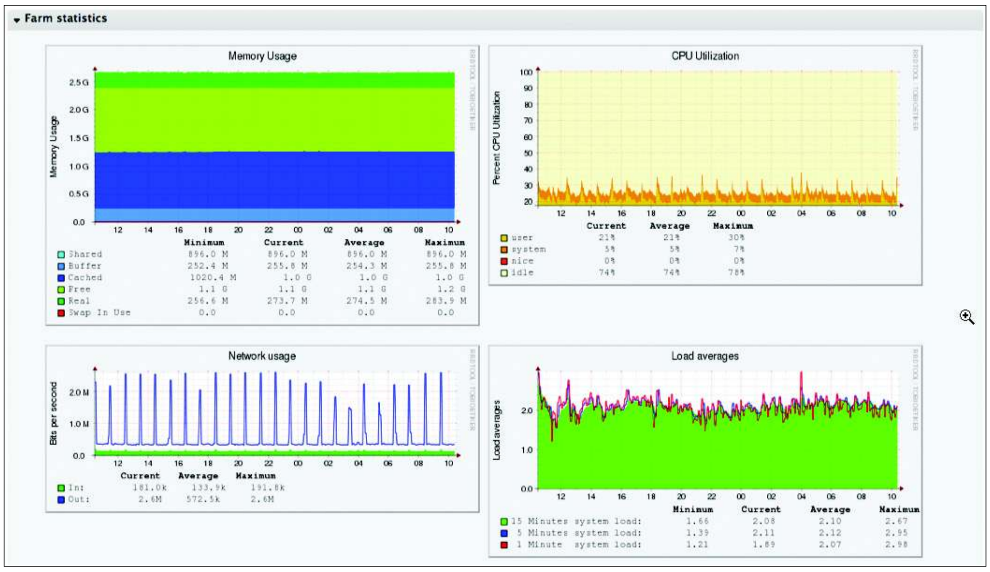
The providers of cloud computing services – Amazon [1], GoGrid [2], Rackspace [3], and Google [4] – currently offer slightly different suites of services. Interoperability is a big issue among those in the cloud ecosystem, because tying your app to a single provider could hurt in the long run; if your scalable app works only on EC2, how do you migrate if (or when) one of the other vendors offers a lower cost?

Listing 1: Working with S3

```
require 'rubygems'
require 'aws/s3'
AWS::S3::Base.establish_connection!(
  :access_key_id     => 'your access key id',
  :secret_access_key => 'your secret access key'
)
```

Listing 2: Building a Message

```
welcome_message = "Welcome, " + work_object.username + " - here is a special message we worked out for you."
welcome_id = "Welcome-" + work_object.username
AWS::S3::S3Object.store(
welcome_id,
welcome_message,
'weclome-messages'
)
```



```
                Minimum     Current     Average     Maximum
Shared          896.0 M     896.0 M     896.0 M     896.0 M
Buffer          252.4 M     255.8 M     254.3 M     255.8 M
Cached         1020.4 M       1.0 G       1.0 G       1.0 G
Free              1.1 G       1.1 G       1.1 G       1.2 G
Real            256.6 M     273.7 M     274.5 M     283.9 M
Swap In Use         0.0         0.0         0.0         0.0
```

```
          Current   Average   Maximum
user        21%       21%       30%
system       5%        5%        7%
nice         0%        0%        0%
idle        74%       74%       78%
```

```
          Current   Average   Maximum
In:       181.0k    133.9k    191.8k
Out:        2.6M    572.5k      2.6M
```

```
                              Minimum   Current   Average   Maximum
15 Minutes system load:         1.66      2.08      2.10      2.67
 5 Minutes system load:         1.39      2.11      2.12      2.95
 1 Minute  system load:         1.21      1.89      2.07      2.98
```

Figure 2: Amazon says its services do the "heavy lifting" so you don't have to. Apps like RightScale and Scalr help manage the details so you can concentrate on the app.

which means you can scale up to around two-million queries for the service before you're charged. Although you might be worried that you will quickly exceed two million queries as soon as your application becomes world famous, this threshold should be enough to get you started. A Ruby gem provides a solution for building SimpleDB into your web application. You will also find bindings for many other languages. For Ruby, start by installing the gem:

```
gem install aws-sdb
```

To install this gem in your Rails app, refer to the full documentation [5].

Once you set up the model, which you can do in one line,

```
class Post ↩
  < ActiveResource::Base
  self.site   = ↩
 "http://localhost:8888"
  self.prefix = ↩
 "/our_website_users/"
end
```

the cool thing about the Rails bindings is that you hardly notice that you are using SimpleDB.

The first parameter, *site*, is the proxy through which Rails accesses SimpleDB, while *prefix* is the SimpleDB domain in which the data is stored. If you decide to host a user model on SimpleDB, it will still look like any other model:

```
user = User.create(
 :username => ↩
 'dan@example.com',
 :favorite_products => ↩
 {2341, 4251, 2567})
user.save
```

So you could easily move your user tables to SimpleDB but keep the product DB sitting in a relational database and then build most of your pages using preemptive caching. This solution puts all of the really hefty services at the front of your website: unlimited EC2 instances, S3 for static files and caching, and SimpleDB for massive tables.

Suppose most of the site is built from pre-cached fragments and you need to retrieve fragments based on the logged in user. If you still had databases running on EC2 instances (e.g., a MySQL cluster), you would still have to manage how this database scales.

Using SimpleDB, you can just throw all the data in and get the user record back:

```
user = User.find(9876)
cached_snippet = ↩
AWS::S3::S3Object.find ↩
'Welcome-' + user.username, ↩
'welcome-messages'
```

To perform the validation, you use the Rails API as usual:

```
user = ↩
User.find(:first, :params => ↩
{ :username => ↩
'dan@example.com', ↩
:password => 'secrets' })
```

SimpleDB is the place to store all those really terrifyingly big tables, instead of spending days optimizing relational structures and building clever caches.

This example illustrates the real benefit of cloud computing services – someone else can do the heavy lifting. SimpleDB, S3, EC2 and the many other services each provide an efficient way to perform an important task.

Scale Up; Scale Down

Once you've created the application that will be using cloud services to scale beautifully, how do you perform the actual scaling?

Part of the "ecosystem" building up around AWS (as well as many other web services) are tools such as RightScale and Scalr that do the work of starting and stopping servers as they are needed (Figure 2). With both systems, what you do is design the kinds of servers you need and then set some scaling rules based on the CPU load, maximum number of machines, or any other relevant consideration. These services talk directly to AWS on your behalf, so you don't have to start and stop using the AWS EC2 API directly.

You can sign-up for either of the services and deploy your application to as many servers as you like. If you really do like "getting under the hood," you can always role your own scaling system that speaks directly to EC2, S3, and other services. The API is based on SOAP, with bindings in most common languages.

Although each system works in its own way, the principles are similar. For example, with the preceding example of a system to generate recommendations for customers, you need at least one server running all the time, but if the number of active customers grows, you might want to increase the number of servers automatically.

Recommendation App

You can create a "recommendation app" that pulls items from the queue and generates recommendations. Along with this app, you can implement a set of rules that creates a new server instance if the CPU goes above a certain level (e.g., 70 percent).

You could also include rules that start a new server based on the number of items in the queue (e.g., if the total number of items goes above 1000, start up another server). This rule set will keep the queue down to a minimum by throwing in additional processing power when the line gets big. Your application is truly and dynamically scalable.

Find Bottlenecks

Building software for cloud computing is largely about optimizing the application given the new tools available to you.

When you need to scale, your first question is: How? Do you have a lot of users, or just a lot of hits? Can users share some data, or is each user's data unique? What *must* happen when users hit the site, and what can you cache in advance?

If you compare how people have solved the problems of scaling common applications, such as WordPress, and online applications, such as Twitter, you'll see that there are very different problems.

Scaling on the cloud involves a lot of decoupling – making one thing work completely independently of another so that each process isn't held up by another. Sometimes, doing this will require re-working your application, but if you're lucky enough to be building from scratch, make sure you don't choose methods out of habit.

If you are scaling an existing application or building one from scratch, you will need to optimize. Optimization is one of the never-ending tasks – Google goes on about how they shave every millisecond off the page load time as possible, and if you've ever had to optimize, you'll understand this.

Make sure you have tools for optimizing and you know how to use them – each language has a host of tools for benchmarking and profiling the code. Optimization is important when you think about scaling, because if your code contains a bottleneck, the speed problem will multiply with the number of users. Plus, bottlenecks are good candidates for decoupling the application.

Building in Clouds

As soon as you worked out the quickest way to deliver a web page from your rack servers, along came cloud computing providers with thousands of servers and a whole new approach. Getting these cloud services into your toolset will save you from much of what Amazon calls the "heavy lifting." ■

INFO

[1]	Amazon Elastic Compute Cloud (EC2): *http://aws.amazon.com/ec2/*
[2]	GoGrid: *http://www.gogrid.com/*
[3]	Rackspace: *http://www.rackspace.com/solutions/cloud_hosting/index.php*
[4]	Google Apps Engine: *http://code.google.com/appengine/*
[5]	"Using SimpleDB and Rails": *http://developer.amazonwebservices.com/connect/entry.jspa?externalID=1242*

UKUUG - *the UK's Unix and Open Systems User Group*

SUMMER TECHNICAL CONFERENCE

'Free and Open Source Software Projects'

Friday 7 August - Sunday 9 August 2009

The Conservatoire, Birmingham City University, B3 3HG

To book and for more info, visit
www.ukuug.org/summer2009

Lots of different Projects are involved this year:
Open Street Map, the Drupal UK User Group.
Exim mail server developers, TeX user group, and many others.
We're running three parallel topic tracks, including:

- **Exim and other e-mail systems**
- **Typesetting and mathematics with TeX and LaTeX**
- **FLOSS languages and programming**
- **Games development**
- **Large system administration**
- **Linux kernel**
- **Open telephony**
- **System and network management**

Special early-early bird student rate of £10 if booked before the end of May.

NEW THIS YEAR *Social Programme for delegates' partners*

CHOICE OF TUTORIALS

- **Using the GIMP, (half day)** ***Neil Woolford*, a professional photographer.**
- **Getting started with Drupal, (full day)** ***Peter Brownell, Robert Castelo***
- **Setting up a VoIP phone system, (full day)** ***Quentin Wright.***

If you'd like to give a talk, send details using the form at: www.ukuug.org/summer2009/register.shtml
Subscribe to the conference mailing list at: http://lists.ukuug.org/mailman/listinfo/summerconf

Event Sponsorship opportunities are available - contact: office@ukuug.org
Event Sponsored by: **IBM, Linux Magazine, Novell, Sun**

UKUUG Ltd. PO Box 37, Buntingford, Herts SG9 9UQ Tel: 01763 273475 Email: office@ukuug.org