Highly Available Websites Despite Massive Failure

JAN GRAJKOWSKI, ZACH SWASEY and PAT WHELAN
{jang115, zswasey, pat717}@vt.edu
Virginia Tech

We introduce a new software project to test highly available websites and attempt to point out weak points in website infrastructures. This paper will cover the goals of our project, as well as the April 21st, 2011 event that sparked the need for such a thing. The software results of the project can be useful for testing how well a website faces massive failure in the cloud. We also discuss recommendations for developing a highly available in the cloud, with ideas not just specific to the cloud but could apply to a regular data center as well.

Categories and Subject Descriptors:

General Terms: High availability, Failure models, Cloud

Additional Key Words and Phrases: high availability, failure models, cloud

1. BACKGROUND

On the 21st of April in 2011 Amazon Web Services (AWS)[1] suffered major outages in its Elastic Block Store (EBS) offerings. EBS is a distributed, replicated block data store that is optimized for consistency and low latency read and write access from Elastic Compute Cloud (EC2) instances. Due to a network change pushed in the US East Region of AWS there was a short network outage and many EBS nodes were isolated from other EBS nodes in their replication clusters. When the network connectivity was restored all these nodes rapidly started looking for server space where they could re-mirror its data. This rapid search by many nodes at once caused roughly 13% of EBS nodes to be in a “stuck” state where they were unable to process normal requests.

Issues related to this outage lasted approximately 2 days as EBS nodes and volumes were restored. In this time, however, many sites were affected and down as a result. A partial list of websites includes: Reddit, Quora, AirBNB, and Kickstarter. A notable exception was Netflix, one of AWS’s largest customers. Netflix doesn’t use EBS itself, but it does use other services that AWS offers that use EBS on the backend. While they should have been affected by the outage Netflix had testing in place that made it so they were able to stay afloat. They had graceful degradation in place all over their infrastructure allowing them to recover from failure in a quick and easy fashion. One of their main solutions was a testing framework known as “Chaos Monkey”.

What Chaos Monkey does is randomly select services that exist in the Netflix infrastructure and terminate them. After this, testing is performed to determine how the website handles the loss of these services. When a service goes down they should be automatically recovered without any manual intervention. What Chaos Monkey wasn’t designed for, however, was the failure of an entire Availability Zone which is essentially what happened on April 21st. Despite this though, Netflix was able to weather the storm of the failure quite well.

[1]http://aws.amazon.com

2. INTRODUCTION

We set out with three goals in mind: write a replacement for the Chaos Monkey framework that would run on the newly developed OpenStack Cloud platform, create a highly available website infrastructure similar to Netflix’s and attack it with Chaos Monkey, and to develop a website to sit on top of the infrastructure and be able to reliably determine its uptime. The outcome of these three objectives working together would be a website that would be able to stand up against failure. We worked in collaboration with Rackspace and used their Rackspace Cloud[2] and alpha release version of OpenStack[3] to develop and test our implementation of the above goals.

This paper will describe the steps we took to attempt each of the three goals. We will cover the process that was used in developing the Chaos Monkey replacement, the highly available infrastructure, and the website that would sit on top of the infrastructure. Unfortunately, in developing our infrastructure for testing purposes we were unable to get it to an automated state on par with what would be acceptable for such a thing. We did, however, come up with many recommendations as to what should be done to avoid the situation that happened to many websites with the AWS outage on April 21st, 2011.

3. NETFLIX: WEATHERING THE STORM

As mentioned in the Background Netflix was one of the few websites hosted in the affected availability zone that wasn’t majorly affected by the service outage. They way they designed their infrastructure for high availability saved them from failure, thanks in part to Chaos Monkey and their Rambo Architecture. Rambo is Netflix’s Active/Active system of having undifferentiated servers that can handle any and all requests. What this means is that if one server is supposed to be handling a specific job, such as acting as a load balancer, and it goes away then any other server can take it’s place while waiting for new servers to be launched.

During the April 21st event Chaos Monkey wasn’t actually as big a help as some might think, but it surely did help. Chaos Monkey wasn’t designed at the time to handle a whole availability zone in AWS going down, but merely just a handful of server instances here and there. So when the servie outage happened the engineers were unsure how things would be handled even though it was tested by Chaos Monkey. Since the event Netflix has increased their team of Monkeys to a “Netflix Simian Army”[Netflix Inc. a], which includes Monkeys to do all sorts of monitoring of services, testing, and terminating of instances.

One affect that Netflix did have was with loss of connections when using AWS’s Elastic Load Balancer (ELB). Since ELB balances across availability zones first, and then across nodes, a service interruption in a single availability zone can cause a bunch of connections to the ELB to not go through. When servers started crashing in the affected availability zone, that portion of the ELB

[2]http://www.rackspacecloud.com
[3]http://www.openstack.org

round robin balancing stopped responding, causing a drop in connections. ELB is also partially backed by EBS, causing even more trouble when the EBS volumes were unable to be contacted.

4. ANARCHO CHIMP: CHAOS MONKEY ON OPENSTACK

Our first goal was to recreate Chaos Monkey in an OpenStack environment. While Netflix only briefly describes Chaos Monkey and what it does, they don't release any source code for it or discuss how it works in relation to AWS. There was, however, a publicly available open source implementation[4] hosted on Github. It was written in .NET and only implemented the very basic idea proposed by Netflix: select random severs without discretion and terminate them. This was a good place to start, and we set out to develop just that. As a fork of the concept of Chaos Monkey, but not any actual code, we decided that a new name would be appropriate, thus Anarcho Chimp was born.

While the implementation of Chaos Monkey on Github was written in .NET we decided to write our implementation in Python. We chose Python because in the server administration world, Python is king. There are also many different libraries available for Python that can do a wide variety of tasks, so there would undoubtedly be a tool available for whatever we need. To talk to the OpenStack API we used the Python bindings developed by Rackspace, known as `python-novaclient`[5]. Originally we had considered developing our implementation of Chaos Monkey as being cloud service independent so that it could run on AWS, Rackspace Cloud, OpenStack, or any other cloud services provider. To this end, we investigated using Apache's Libcloud[6], but found that it abstracted away too many of Rackspace and OpenStack specific API calls. The next idea was to abstract out the platform specific parts of our implementation so that the python-novaclient or Boto (python bindings for various AWS APIs) specific sections could be dynamically loaded for attacks against OpenStack or AWS, respectively. This was abandoned due to our main goal being to develop this system for OpenStack specifically. However, it wouldn't be too difficult to refactor the codebase to allow for the latter description in the future.

With the programming language and API bindings settled upon we set out to design our Chaos Monkey implementation in a way that could be expanded upon in the future if needed. This was pretty easy in Python, as it has full object oriented capabilities. In the process of designing the program we decided that it should do more than just randomly kill instances, but it should also attempt to be smarter about it. Following brainstorming and meeting with our professor we came up with four basic failure models that we think are a good starting point.

4.1 Failure Models

Our Anarcho Chimp includes several failure models: Random Failure, Network Failure, Process Failure, and Graph Failure. These can unleashed upon a random subset of all instances, or a random subset of instances that are tagged with a specific string of metainformation supplied at runtime.

4.1.1 *Random Failure.* This is the classic model described in Netflix's description of Chaos Monkey. It gets supplied the list of instances to be considered, which consists of all instances or the subset of instances with the specified tag, and a number of times to repeat. Given these two items, it will randomly select an instance and call its delete method. That server is essentially removed from the list of considered servers, and the process is repeated a specified number of times.

4.1.2 *Network Failure.* This is the next model that we wished to implement for Anarcho Chimp. What it does is simulate a network outage on the current server instance. In AWS there's a concept known as "Security Groups" which are basically `iptables` rules that get applied to a server instance. Using security groups in AWS to simulate a network failure would be easy, since we would need only remove all security groups from the selected instance, and add our own security groups that will drop all incoming and outgoing packets to the instance. OpenStack, as of the Diablo alpha release, doesn't have support yet for security groups, so we had to develop a different way to accomplish the same thing.

Since security groups would basically act as `iptables` rule, we decided that we might as well install the actual `iptables` rules to do just what we wanted. To accomplish this we would need to be able to `ssh` into the instance and run our commands. This would require no additional information at Anarcho Chimp runtime if the server instances were given the runner's public key at boot time. If this isn't the case, a JSON file can be passed in at runtime tying a server IDs to their respective root passwords. At this point, the process is very similar to Random Failure: the model is given a list of instances to act on, one is randomly selected, and an action is performed upon it. Using the python module Paramiko an `ssh` session is established to the server and the following command is issued:

```
nohup sleep 5 && iptables -P INPUT DROP
&& iptables -P OUTPUT DROP && iptables
-P FORWARD DROP &
```

This will install the iptables rules to drop all incoming, outgoing, and forwarded packets. It will also sleep 5 seconds in nohup background mode, so that the `ssh` connection has time to end without getting stuck.

4.1.3 *Process Failure.* In this next failure model we wanted the ability to not just kill a random server instance, but also random processes running on that server. Examples of such processes are: mysqld, httpd, haproxy, mongrel. These are the main processes running on servers that create the software infrastructure for the website. The process in developing this model is very much the same as for Network Outage, but when we `ssh` into the server we're calling killall to kill the process. This model requires that a JSON file gets passed in at runtime that contains a mapping of server ID to a list of main processes running on that server. The model will then act much like Random Failure, and select a random instance and then a random process, establish an `ssh` connection, and kill it.

4.1.4 *Graph Failure.* The final model that we implemented depends on a user supplied graph structure of the infrastructure. It will then determine which instances in the graph have the highest degree, and terminate those. This simulates the most connected server in a website's infrastructure crashing. The format supplied is JSON with a list of servers each with in and out connections specified to other server IDs.

4.2 Further Development

As stated earlier we originally wanted Anarcho Chimp to be platform independent and to be able to attack any cloud services

[4]https://github.com/simonmunro/ChaosMonkey
[5]https://github.com/rackspace/python-novaclient
[6]http://libcloud.apache.org

provider. This would be a main feature in a future iteration of the software, but it would be pretty easy to accomplish and integrate with Boto or other python API bindings. We would also like to develop further failure models that simulate various types of failures that might affect a server, such as hardware failure. Along these lines, we'd like to implement several features of some of the other works in Netflix's Simian Army[Netflix Inc. a], including processes that simulate delayed calls to APIs, killing instances that are no longer healthy, or killing instances that don't seem to be doing anything.

5. WEBSITE INFRASTRUCTURE IN THE CLOUD

After writing an implementation of Netflix's Chaos Monkey on OpenStack, we needed something to test it against. We set out to develop a highly available infrastructure that would run on top of OpenStack and be resilient to the failure models implemented in Anarcho Chimp. From the beginning we had a few requirements in mind: complete automation so servers could be reinitialized without human intervention, load balanced so that the loss of a single server in a replicated cluster would be unnoticeable, highly replicated so that no data would be lost in the event of a failure, and no single point of failure. Each requirement will be talked about in depth, covering methods that we considered and eventually tested, problems we encountered, and solutions to those problems.

5.1 Automation

We knew that there would be a need for some sort of configuration management tool, such as Puppet[7] or Chef[8], to be able to bootstrap new servers and make sure they were up to date from time to time. Requirements that we had for such a tool was that it had to be able to keep various templates for different server types, keep up to date versions of files that would need to go to different servers, and allow the bootstrapping and integration of new servers with the current state of the infrastructure. Puppet was chosen due to already being somewhat familiar with it, and its fulfilment of our requirements.

Setting up Puppet was relatively easy. We were able to create some templates for various server types, such as "webserver", "dbserver", and "webhaproxy", and say exactly what we want done on each. For example, in the webserver config type we would want it to install Apache, start it, and make sure that it's always running. On the `puppetd`'s first run on the client it will grab the configuration and make sure that all requirements are met, and if they aren't it will take the steps to ensure that they are. Puppet also allows us to have a central repository of files that all servers might need, such as the master hosts file that includes the IP addresses and hostnames of all servers in the infrastructure. This is useful so that servers will be able to talk to each other as "dbserver1" instead of having to hardcode an IP address into configuration files.

New servers are created from a blank server image created with modifications to include an updated operating system, updated `apt` repositories, Puppet already installed, the `puppetmaster` hostname already in /etc/hosts, and the `puppetd` cronjob running every minute to poll the `puppetmaster` for config changes. Once a server is initially bootstrapped the only changes that'll be happening to configs is updating /etc/hosts when new servers go up and down.

Puppet by itself doesn't have the ability to do everything we needed in regards to automation. So we set out to write a simple script, which was dubbed Ventriloquist, that would run every minute and check on the current state of the infrastructure. If a server had somehow died, it would spin up a new one of the same configuration, and tell Puppet that it needs to be bootstrapped when it first checks in. This script that ran on a cronjob read in a description of the infrastructure in JSON format that said what should be running and what type of server it should be; if the current state didn't match, it was made to match. The process from start to end goes like this:

(1) Ventriloquist on a cronjob every minute does a request to get a list of all server instances.
(2) It then compares the names of the instances to a JSON file that provides the whole infrastructure, including server name and what type of puppet configuration they should be.
(3) If a server is in the JSON file, but not in the results from the API request, then it will need to be relaunched.
(4) A boot request is made with our custom image.
(5) The information for this newly booted image is stored in the JSON file on the desk.
(6) A new Puppet node file is created with the config information provided in the JSON file.
(7) Puppet is informed to clean any certificates previously under the hostname of the new server, and accept the certificate of the new server.
(8) Puppet then waits for the server to bootstrap, and sets the JSON file to reflect this final state.

Our current recommendation is to also have your `puppetmaster` server running out of the cloud and in your own domain on your own hardware. This will mediate any issues of the configuration server being the victim of a data center crash.

Unfortunately, we were unable to get automation working flawlessly the way we wanted. It wasn't an inherent problem with Puppet, but in getting the servers bootstrapped and reinitialized in a the correct manner. For example, when starting a new dbserver it needs to be brought up to speed with the current state of the database, users have to be created, and it needs to be set in its rightful place in the database hierarchy (is it a master? a slave?). These tasks were difficult to get right, and due to time constraints with the project were left unfinished.

5.2 Load Balancing

For websites nowadays load balancing is a key part of making a highly available website. Instead of having one server take on all the work, it makes more sense to spread the work out amongst many less powerful servers. One popular use is to load balance over several identical web servers, all running the same software. When a request comes in to the load balancer it will route it to the server that it feels would be best to get the next request. This causes the work to be evenly spread out amongst all the webservers. The load balancer will periodically check the health of a server that it's managing to make sure that it's able to still handle requests. In the case of web servers it will do a health check on port 80 to make sure it's still listening, and if it isn't over a particular time span then it will be removed. Once a server is removed from the load balancing its traffic is redistributed over the remaining servers.

There are several options for load balancing software to operate in the cloud. AWS has their own load balancer called Elastic Load Balancer (ELB), that has many configuration options and APIs available to make it highly useable. However, during the April 21st event it had a high number of failures because parts of the

[7]http://www.puppetlabs.com
[8]http://www.opscode.com/chef

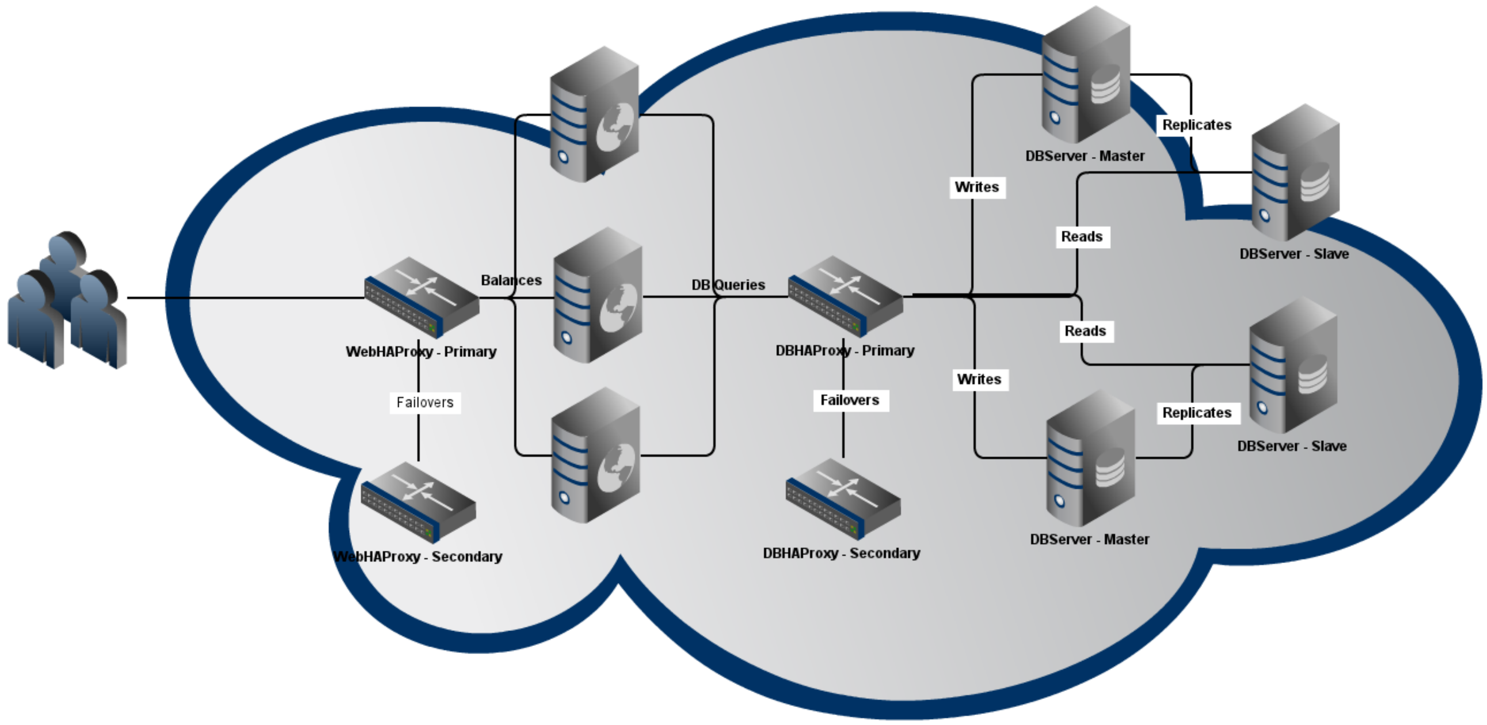

Fig. 1. The highly available website that we proposed. Not shown is the offsite puppetmaster config server.

ELB backend utilize EBS volumes. RackSpace Cloud offers a basic load balancer as a service, but it doesn't have more than the most basic configuration options. Third party software includes: HAProxy, Linux Virtual Server, and Ultra Monkey. HAProxy[9] is widely used for highly available load balancing and provides many configuration options for basically any setup. Linux Virtual Server (LVS)[10] is a much more advanced option that can load balance amongst servers within a Linux cluster, amongst applications on those servers, and provide virtual IP services. Ultra Monkey[11] is a collection of various softwares that provides configuration for them, including LVS, ldirectord, and the Linux-HA software heartbeat.

For the purposes of our small testing infrastructure it would seem that Ultra Monkey would have provided the best solution to most everything we would need for load balanced and highly available services. Unfortunately it is no longer actively developed, and the last release was 6 years ago. After attempting to get LVS working, and failing due to the complexity and necessary overhead, we settled on using HAProxy as our load balancing solution. It was a very easy setup that consisted of merely a single configuration file that contains the necessary options about which servers should be load balanced, what port they should be listening on, what port HAProxy should be listening on, and how the load should be balanced. The default balance algorithm is round robin, which fairly distributes the load amongst the servers in turn. At this point, it was merely a task of creating a config in Puppet for a "webhaproxy" type, ensure that it would install `haproxy` and keep it running, and load it with a configuration file that contained the list of webservers.

5.3 Data Replication

The idea of data replication was only necessary in our concept for our database servers. We wanted new database servers to be able to start off where the old one that it's replacing ended. To accomplish this there needs to be data replication amongst the various database servers. Methods for data replication for MySQL[12], our database of choice, exist in two forms: database level, and block level. Database level options are provided by MySQL's built in replication services. This is a common method that allows a database to act as a master, receive all writes, and replicate the written data down to slaves. These slaves then receive all the reads that would happen against the database. This is a nice method, but bringing new servers up to date requires dumping the current database and importing it into a new slave, and having it catch up to the current state of the master. This can be time consuming and error prone, as MySQL level replication often is.

Block level replication often comes in the form of Distributed Replicated Block Device (DRBD)[13] which basically implements a RAID1 setup over a network. This means that a block device on each server can be kept mirrored between itself and another server. DRBD is nice because in newer versions it can mimic RAID1 between three servers, with the first two acting as primary and secondary, and the third server acting as disaster recovery. Without a

[9]http://haproxy.1wt.eu/
[10]http://www.linuxvirtualserver.org
[11]http://ultramonkey.org
[12]http://www.mysql.com
[13]http://www.drbd.org

third server, DRBD would have to catch up the block device on a newly launched server, which could take a while. However, with this third server waiting around with the fresh set of data it just needs to failover to it. Then a new server can be launched with and get caught up in the background, acting as the new disaster recovery DRBD node.

Given the capabilities of DRBD versus MySQL replication, we recommend using the former. It's a bit more complicated to set up, but the gains are well worth it. As mentioned earlier, we were unable to get automation set up reliably for our database servers, which includes the data replication. Although we are recommending DRBD, we felt the configuration necessary and the risks to our small demo was too much, so we opted to attempt the easier MySQL replication route.

To make the database highly available and up to date we decided to use HAProxy, the load balancer discussed earlier with regards to web servers, along with servers in a Master-Master configuration, and servers in Master-Slave configurations. Two servers would be in Master-Master configuration, both being able to be written to and update the other. Each of those servers would have a legitimate Slave attached to it that would be kept up to date by having updates pushed to it from its Master. These slaves would only be used for reading data. HAProxy would be sitting in front of this setup, with configuration setup to provide a port to listen on for writes, and it would direct all traffic to be balanced between the two Masters, and another port to listen on for reads, which would balance reads between the slaves. When new servers come up and down by Puppet they would need to be placed in their appropriate position and set as a Master or a Slave.

5.4 No Single Point of Failure

A single point of failure is just what it sounds like: one single server that if it were to crash, a service might fail. This is at the heart of the reason behind high availability. One of the main ways to provide no single point of failure is with IP failover. If you have a single forward facing IP address for the web server load balancers, then if that server goes it can switch the IP address over to another server that'll act exactly the same as the original one. Two pieces of software are widely used for this: heartbeat and keepalived. Heartbeat[14] provides a lot of different capabilities to monitor various services, such as apache and mysql, if the service goes down it'll switch over the virtual IP address. Keepalived[15] does basically the same thing, but more barebones, and for our purposes, much easier to set up.

Both require that they're able to listen for the other instance, but this isn't very easy to do in the cloud which lacks multicast support. We had to use the keepalived-unicast[16] patch to allow to specify the secondary server to listen to. This wasn't very difficult, and Puppet handled setting up the configuration files. We would be using `keepalived` only on our HAProxy boxes, so if the primary goes down the virtual IP would be switched out very quickly.

OpenStack provides the capabilities for what they call "floating IPs", but they aren't implemented as of the Diablo alpha release. AWS provides what they call Elastic IPs which are IPs that can be assigned to any instance at any point in time; they are perfect for this situation. Rackspace Cloud allows sharing of IP addresses between server instances in a single IP "group", but only if the initial server with that address is kept alive, which obviously causes problems for our testing purposes. There's also the ability to request an additional public IP address, however it has to be tied to a server at any point in time, which again causes problems for our purposes.

[14]http://www.linux-ha.org/wiki/Heartbeat

[15]http://http://www.keepalived.org

[16]http://1wt.eu/keepalived

6. FINAL RESULTS

Despite the inability to get automation and replication setup in our test infrastructure, we still have a number of recommendations for highly available websites. Figure 1 shows the setup that we had planned on supporting and testing Anarcho Chimp against. As you can see, the frontend WebHAProxy Primary is the only thing visible to the user, but there's plenty going on behind it to make sure the website is always running.

Some of our final recommendations are listed below. They are used by many websites, including Netflix, in keeping their websites highly available and ready for most kinds of failures.

—Keep your `puppetmaster` in your own domain. Then you have much more control over the software and hardware that's handling the automation of your entire website's infrastructure.

—Use DRBD with a third server serving as a hot backup for disaster recovery.

—Use Monit[17] to monitor processes on servers. It can then be used to send an email, or any other kind of alert to warn you that a process might have tied. It will also restart that process if you tell it to.

—Have some sort of IP failover, either by Heartbeat and Pacemaker[18], or keepalived. If using AWS, request additional Elastic IPs. If using Rackspace Cloud, have your HAProxy servers in a shared IP group.

—Use a configuration management tool such as Puppet or Chef. They both provide many of the same features and have Domain Specific Languages (DSL) for their templates and node files, so it's easy to write what you want them to do.

—Use Nagios[19] to monitor everything about your server instances, including: processes, uptime, in/out network traffic, CPU load, etc. Nagios can be integrated with Puppet to provide the ability to use Nagios stats with Puppet configs.

—Consider having N+1 server instances ready just in case something goes awry. In AWS you can purchase reserve instances that are ready whenever you need them and will always be available to you.

—Also consider an Active-Active arrangement of server instances. This implies that any instance can pick up where another left off if it were to crash. Netflix uses a similar architecture codenamed Rambo, where all servers run the same processes and can handle any role in the infrastructure. This allows you to quickly provision these servers.

—Have instances across availability zones and regions, so the website can still be available even if an entire region's data center were to go down, such as an event in 2010 when an AWS data center in Virginia lost power.

7. TESTING APPLICATION

Once we had the infrastructure for the website ready to go, the next step would have been to deploy an actual website on to the web servers and benchmark it. We had developed a Twitter clone alongside the infrastructure in a framework called Grails[20], which

[17]http://mmonit.com/monit

[18]http://www.clusterlabs.org

[19]http://www.nagios.org

[20]http://grails.org

tries to provide an easy Model-View-Controller framework for the Groovy[21] much the same way that Ruby on Rails[22] does.

The application would run on Apache Tomcat[23] on the web servers and be routed to by HAProxy. It was designed so that it'd be able to act as a regular third party piece of software that anybody might happen to run on such an infrastructure. It makes database queries to the the database specified which would be the HAProxy sitting in front of the databases, which would then decide which database server to route the requests to.

If we had gotten the infrastructure to an automated point, we would have deployed this application and launched Anarcho Chimp to knock out a few servers and determine whether or not the site was still fully functional. If it wasn't fully functional we would have been timing how long it took for everything to get back to a normal state. We would also be testing how long it takes pages to load and how the infrastructure holds up against thousands of simultaneous connections using Apache Bench (ab)[24].

8. SOME PRELIMINARY NUMBERS

While we don't have many hard numbers, we can make some estimates. On the RackSpace Cloud the average time for a server to go from the initial boot request to Active state is around 3 minutes. Then, if you run `puppet` on a cronjob every minute or two, you have to wait for that to bootstrap your server. If the cronjob script that checks if the servers are up or not runs every minute, then we can estimate that to get a server back up if the original were to disappear, it could take from 6-8 minutes if everything was automated. In our example infrastructure, this wouldn't affect downtime since everything has a backup and is load balanced. However, if both HAProxy servers go down, there's going to be a problem. We would have to wait the 6-8 minutes for it to come back online and the site would be down in that time.

With regards to HAProxy, it takes 3 failed health checks in a row by default for a server to be considered down. So a server is removed from HAProxy within 10 seconds of it going away. In that time, HAProxy could still route requests to it, so an optimistic loss of requests is around 10 seconds for requests to a HAProxy with a missing server.

9. CONCLUSIONS

Obviously, your mileage my vary. Not every website has the same needs and some site infrastructures might look completely different or implement different ideas for high availability. What we have presented, however, is a good start for a medium sized website that wants to ensure that its services available as much as possible. Also take what we learned to heart: automating stuff like this to always work without any sort of human intervention is difficult. It requires a lot of tweaking and testing to make sure that when the engineers are sleeping that the website will take a licking and keep on kicking.

ACKNOWLEDGMENTS

We'd like to thank Gabe Westmaas at Rackspace Cloud for giving us an account with free range to do what we pleased with it. It was very useful and educational to help learn what Rackspace Cloud and OpenStack are like. Hopefully OpenStack progresses more and becomes a viable option for cloud services compared to AWS.

We'd also like to thank our professor Ali R Butt for many ideas in group meetings through the semester and giving us several extra chances.

REFERENCES

ENRIGHT, G. Easy web server load-balancing with haproy. http://blog.rimuhosting.com/2011/07/05/easy-web-serve-load-balancing-with-haproxy/.

NETFLIX INC. 5 lessons weve learned using aws. http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html.

NETFLIX INC. Lessons netflix learned from the aws outage. http://techblog.netflix.com/2011/04/lessons-netflix-learned-from-aws-outage.html.

NETFLIX INC. The netflix simian army. http://techblog.netflix.com/2011/07/netflix-simian-army.html.

WILLIAMS, A. Using haproy for mysql failover and redundancy. http://www.alexwilliams.ca/blog/2009/08/10/using-haproxy-for-mysql-failover-and-redundancy/.

[21] http://groovy.codehaus.org
[22] http://www.rubyonrails.org
[23] http://tomcat.apache.org
[24] http://httpd.apache.org/docs/2.0/programs/ab.html