Scaling on EC2 in a fast-paced environment

Practice and Experience Report

LISA 11

Nicolas Brousse, Lead Operations Engineer, TubeMogul, Inc.
Email: nicolas@TubeMogul.com

***Abstract* — Managing a server infrastructure in a fast-paced environment like a start-up is challenging. You have little time for provisioning, testing and planning but still you need to prepare for scaling when your product reaches the tipping point. Amazon EC2 is one of the cloud providers that we experimented with while growing our infrastructure from 20 servers to 500 servers. In this paper we will go over the pros and cons of managing EC2 instances with a mix of Bind, LDAP, SimpleDB and Python scripts; how we kept a smooth working process by using NFS, auto-mount and shell-scripting; why we switched from managing our instances based on tailor-made AMI/Shell-scripting to the official Ubuntu AMI, Cloud-init and puppet; and finally, we will go over some rules we had to follow carefully to be able to handle billions of daily non-static http request across multiple Amazon EC2 regions.**

Index Terms - Amazon EC2, scalability, fault-tTubeMogulolerance, infrastructure, DevOps.

I. What is Amazon EC2 and how does it work?

Amazon AWS[1] provide a wide range of web-services. Amazon EC2[2] is part of AWS as a public cloud solution. EC2 let you start servers, called instances[3], on-demand. You are billed per-hour of usage and can stop an instance at any time. You can start your instance in a given geographic Region and Availability Zone[4].

Because of the large adoption of EC2, Amazon added a layer of indirection so that each AWS account's Availability Zones can map to different physical data center equivalents[5].

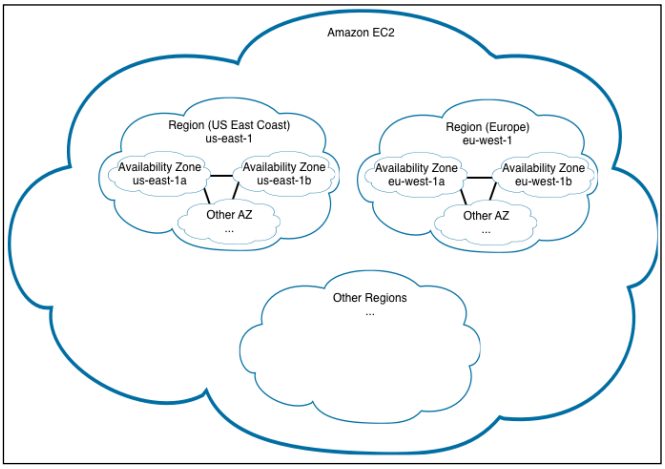

Fig 1. Amazon EC2 : Region and Availability Zone

When starting an instance, you will generally have to provide at least four pieces of information: the AMI[6] (server image), the instance type[7] (ram/CPU/arch), the Security Group[8] (firewall rules) and the Availability Zone. You can start an instance by using the Amazon EC2 API or the web console. By default an Amazon instance is started with some defined ephemeral storage space. Any data on it will be lost if you stop the instance. To use permanent storage you need to use solution like EBS. When stopping a server you lose the attached public and private IP. A new instance will have different IPs. The only way to keep a public static IP is to use Amazon EIP[9].

In September 2010, Amazon introduced some important features: Tagging, Filtering, Import Key Pair, and Idempotency. By adding customized tags (like hostname or profile name) you can easily filter your instances or EBS[10] volumes based on the given tags. In short, tagging and filtering lets you manage your own meta-information for each Amazon cloud resources.

II. Keep some order in your cloud

There are many client bindings built for the Amazon EC2 API which make it quite easy to use and implement. We started to use EC2 in 2008 by taking advantage of the computing ability that Amazon provide. We start a few dozen of servers for a few hours a day to fetch and aggregate data from different partners. The aggregated data are pushed into our shared MySQL cluster at our Colo center.

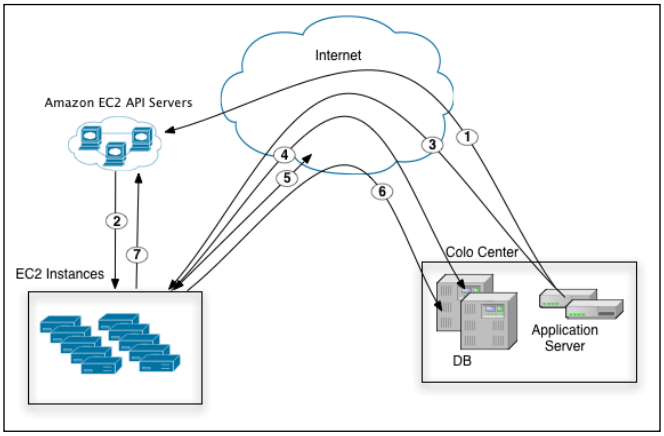

Fig 2. EC2 and Colo center

In Figure 2, you can see how we interact with EC2 to crawl our partners API and store data in our database. 1) our

application server calls the Amazon API at defined interval to start Amazon instances. 2) Amazon launch the instances we requested. 3) we push our code to the EC2 instances and start our program. 4) our application open an SSH tunnel to our databases. 5) we crawl our partner's API and aggregate the data as we want. 6) we write the results to our databases. 7) EC2 instances kill them-selves when they are done crawling.

This design works great and requires really low maintenance. Though, when you work in a startup environment, product evolve quickly. We needed to quickly develop our new video analytic product with a large number of servers to handle the analytics for billions of video stream per month. We chose to build this new product entirely on EC2. This let us to change the application quickly while the product grew without worrying about adding servers, rack, wiring, etc. Because of the nature of our product, we needed permanent storage, that's why we started to use EBS volumes.

To be able to add or remove nodes easily with different instance profiles it's important to be able to quickly identify what a server is doing and identify what its role is (Web server, Database, Hadoop namenode/datanode, etc.). To keep some order in our cloud we used clear security group, human readable hostnames (no ip-XXX.compute.internal or domU-XXX.compute.internal), NFS home directories and a strong and flexible monitoring.

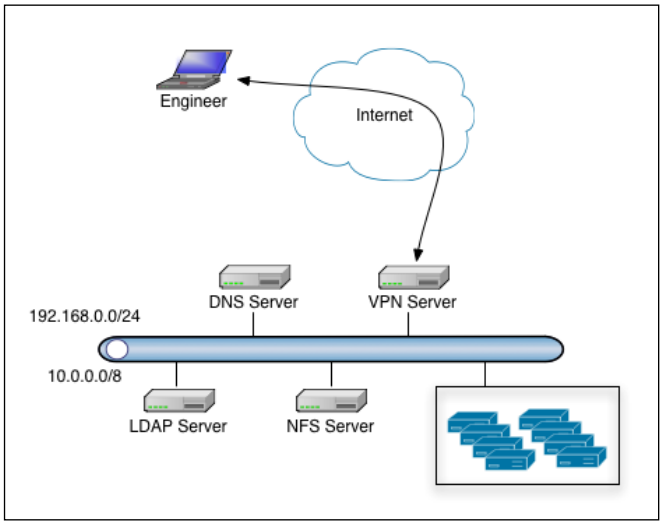

Fig 3. EC2 and our private network

A. Controlling access to the servers

1) Amazon EC2 Security Groups can get a bit cumbersome to manage especially when you want to access servers from anywhere without updating your rules while keeping a strong security policy. It's easy to forget to update or remove an old ip, etc. This is why we chose to manage our servers by setting up OpenVPN[11] servers on two of our Amazon instance using static IP, aka EIP. The ingress rules for our Security Groups stay simple by allowing SSH only from those VPN servers and by opening only the required public port if any. The VPN (using OpenVPN with auth-ldap[12] plugin) add another layer of security ensuring that only people with a valid username and password and a valid unique certificate can get access.

2) In addition to firewalls, we needed to give restricted access to some DBA, developers or contractor. Some needed root access. Our rule of thumb: *"You only get the permission you really need"*. No need to give root access to every server to your boss if he don't even know what to do with it. To manage those permissions and user accounts we used OpenLDAP[13]. All our instances are configured with pam_ldap. We extensively use pam_filters to grant access based on hostname, host group and Availability Zone.

```
pam_filter |(host=dev-mysql01.us-east-1b)(host=dev-mysql01.us-east-1)(host=dev-mysql01.\*)(host=dev-mysql\*.us-east-1b)(host=dev-mysql\*.us-east-1)(host=dev-mysql\*.\*)(host=\*.us-east-1b)(host=\*.us-east-1)(host=\*)
```

At any time we can grant or revoke access to any users for a server or multiple servers in one or multiple regions.

B. Identify running instances

Having obscure hostnames doesn't make your life easy when you start to deal with multiple instance profiles and multiple products with an extra-small sysop team (one or two people). When a product is in its early days with frequent changes, developers often needed access to the servers to be able to troubleshoot issues and find out why their last release wasn't working as expected. To help identify our hosts we used one of our EC2 instances as a management server configured with a DNS service (Bind[14]) patched for the ldap backend[15] and a LDAP service (OpenLDAP 2.4) using some of our own LDAP schema. For each host we stored in LDAP the private IP (10.0.0.0/8) and the public IP (it can be an EIP). Each host that we started used an AMI configured with the given private IP of the name server. Our resolv.conf would look like this:

```
domain <product>.private
search <product>.private <product>.public
nameserver 10.X.X.X
```

When starting an instance we also used the user-data to update the /etc/hostname. The user-data is an optional parameter you can use when starting an EC2 instance. This can support up to 16KB data. On the server you can fetch those user data at boot through an init script doing a curl command:

curl -s http://169.254.169.254/latest/user-data

From there, a lot become possible. In our case, we initially used the user-data just to pass our server hostname, example: "hostname=dev-mysql01". Note that, in the same way you can have access to many meta-data of your running instance:

curl -s http://169.254.169.254/latest/meta-data/

The pam ldap was configured to use the DNS entry to get the LDAP server IP.

uri ldaps://ldap.<product>.private

We started instances using a Java command line tool, called ec2ldap. We wrote it using Typica[16] (Java Binding for Amazon API), SQLite[17] and LDAP. We kept tracking of all our instances name and profiles in a SQLite database and used a script called Cerveza wrote in Tcl/Tk to access our hosts easily and do large maintenance with some one-liners:

./cerveza remote mysql[1-40] service mysql restart

With the SQLite database and Cerveza, it was easy for us to run over all our EC2 instances and update the resolv.conf if our management box went down and got a new IP. This worked well for a while but there were some important single point of failures[18] (SPOF) that finally bit us.

C. The benefit of NFS auto-mounted home directory

As stated earlier, developers needed easy access to the servers. To make their life easier we did setup an NFS export on our management box and used Autofs to mount the home directories on all our EC2 instances.

```
/etc/auto.master:
        /home   /etc/auto.home  intr,soft

/etc/auto.home:
        * fstype=nolock,noatime,soft,intr nfs.<zone>.private:/
home/&
```

This setup makes it easy to run a script across multiple instances without copying the instance to each host. It has been a great help in our dev environment but also when troubleshooting many servers in production. It's convenient, because you get your bash aliases or user script everywhere you login, etc. Unfortunately there is a downside, your access files can get slow, home dir can get stuck or permanently mounted if a service write to the home directory or keep a file descriptor open, etc.

In many cases we ended up using those auto-mounted home directories to run shared scripts on the first boot of an instance to deploy code, build our Raid devices with multiple EBS or reassemble them using mdadm or LVM.

D. Instance monitoring with Ganglia[19] and Nagios[20]

We choose to monitor our infrastructure with Nagios and Ganglia. It was a no-brainer for Nagios as we already used it to monitor our Colo servers and were quite used to its configuration. Ganglia was new for us as we used to graph our servers with Munin[21]. In our case, the decision between Munin and Ganglia was made on poll versus push model. Munin server poll each client, this requiring many resources on the main server especially when building each graphs. Ganglia uses a push model, each client report to the main process (gmond). Ganglia allow much more flexibility in graphing grids and clusters although we couldn't use the multicast support. For security purposes, Amazon EC2 doesn't let you to do multicast (or broadcast) on their network.

We configured multiple gmond processes on our management box to listen on different ports and collect data in different cluster group (one per Amazon Security Group) then just one gmetad process to collect all the data from each local gmond. This helped us to organize our graphs. Our EC2 instance were getting configured at first boot by running a ganglia configuration script that ensures the instance reports to the correct gmond process (if instance in SG dev, reports to port 8630, if SG mysql, report to 8631, etc.). Ganglia is a powerful solution so we were able to use the Python module to graph[22] our Java process using JMX[23] with JPype[24]. All those data are grouped in different dashboard and give us a quick way to spot issues.

For our Monitoring we use Nagios 3.2 with NSCA[25] and regex (in nagios.cfg: use_regexp_matching=1). We defined some generic service definitions for each cluster of servers. Some of our checks were directly looking at our RRD[26] data generated by Ganglia. Because of the quickly growing numbers of servers and services monitored we started to have too much I/O (read/write RRD files). We started to use rrdcached[27] which solved most of the problem but we still had many Nagios active checks which occasionally lead to swapping or slowness during checks. To fix the problem we simply split our ganglia load between two different management boxes, both servers use rrdcached to reduce IOs.

III. Learning the hard way

(or how to lock yourself out of your servers...)

While we were building our infrastructure and upgrading our network configuration, we were aware of few SPOF being introduced but they had a low impact or no impact on our production environment. However, what was initially designed for convenience and laziness became critical. The way we started to depend on those services make them even more critical. We didn't see it coming initially. This is the story of a three days nightmare starting with a VPN outage, then NFS/LDAP outage locking us out of all our EC2 instances.

A. The outage

1) For some reason, our file system storing our Nagios and Ganglia files were corrupted (EBS or Raid problem). This lead to many process getting stuck trying to access the faulty device. Too many resources were being used so the OOM Killer started killing processes, including our VPN process. After many reboots of the management server, nothing came back up. The console output showed a prompt for fsck check due to the faulty device. We had to kill the instance and start a new one.

2) The new instance failed to start. It prompted us again for fsck on our EBS volumes (used for NFS home dir). In fact, the mount point was defined in the fstab in the AMI, so it kept trying to mount the failing EBS with no way for us to fix it. There is no KVM with EC2, so we didn't have any way to try to recover from this situation. We ended up starting a new instance with an old AMI from which we removed the fstab so we could start the instance and finish it manually by running fsck, etc.

3) After reboot, our instance got a new Private IP allocated. This meant a new IP for our DNS, LDAP Producer and NFS. After recovering our instance we reimported our last ldif backup to LDAP. As the DNS server IP was hardcoded in our instance, we had to "manually" login on each server using a local account with the ssh keypair then update the resolv.conf, dnsmasq.conf, dhclient.conf, restart autofs and dhclient.

4) Unfortunately, as we used an old AMI for our management box, we lost many configuration settings breaking our Nagios and Ganglia services but also our command line tool (Cerveza) used to query our SQLite DB and easily access any hosts. This slowed our ability to recover a basic setup to be able to see what was wrong and fix it.

5) The ssh backdoor didn't always worked. We had to restart many instances manually. At boot they couldn't load our boot scripts from NFS. We had to login and finish the boot process manually by fixing Autofs then run the boot scripts. We also had to reconfigure many ssh tunnels, fix mysql replication, and recover missing or outdated configuration files, etc.

6) Some of the servers were using private IP in the EC2 Security Group, rebooting those server make the outage more complex as we needed to review all our security rules.

Luckily, this outage didn't affect our production services but it did lock us out of our servers for a long time. Needles to say, we took some time to revisit what went wrong and how we can fix it.

B. What we quickly fixed

1) One of the biggest pains during this outage, was our pam ldap and ssh configuration. Long timeout was preventing us from login into many servers (the cumul of timeout were higher than our ssh LoginGraceTime timeout, set to 2 min.), so the first thing was to reduce the autofs and ldap timeout and change nsswitch to look at the local account before ldap so even if our dns and ldap goes down, we still have an ssh backdoor to login and do local fix or maintenance.

```
/etc/auto.master :
   /home        /etc/auto.home timeout=5,retry=0,rw,intr,soft

/etc/nsswitch.conf:
   passwd:    files ldap
   shadow:    files ldap
   group:     files ldap

/etc/ldap.conf:
   timelimit 15
   bind_timelimit 5
```

2) We fixed our resolv.conf to handle better failover using: options attempts:1 timeout:1
3) We set up a better service and dns caching on each host using nscd instead of dnsmasq. We enabled caching for group, passwd, hosts and services.
4) We configured a secondary VPN service on our second management server and configured the OpenVPN clients to use "remote-random" option.
5) We stopped saving our fstab in the AMI so we could boot our instance even when a fsck is required.
6) We stopped using private IPs in our EC2 security group
7) We use a Haproxy[28] loadbalancer for DNS and LDAP service via Public IP using EIP.
8) Better version control of our boot scripts and AMI. We now manage almost everything with our configuration management tool.

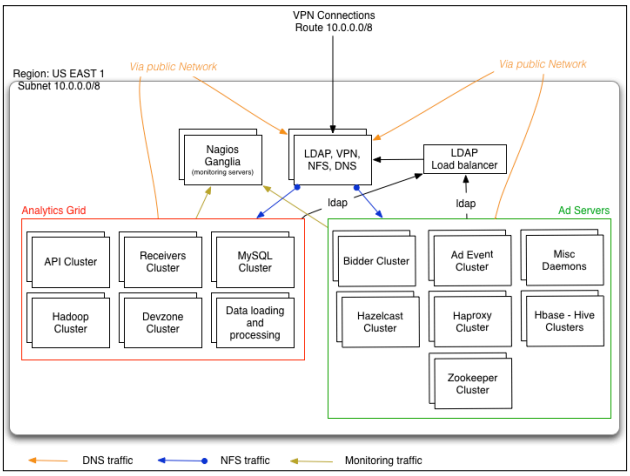

Fig 4. Network Flow between Clusters and Grid

IV. Going Worlwide

While our business evolved, we had a need to have a presence in different part of the world. This is easy to do with Amazon multiple region, though we have response time constraint with many partners. Our ninety-ninth percentile response time must be under 120 ms, including network round trip. Our partners are within 60 ms of our Amazon servers so it doesn't leave us much room especially if you consider the network variation inside Amazon's network or a noisy neighbor.

While building our international clusters, we tried to keep two goals in mind. First, how to reuse our existing tools and automate as much as we can. Second, do not create new SPOF failures in one region that would impact the others.

A. Simplify the instance boot process

With over 500 EC2 instances spread in multiple regions, we had to make our life easier. We got rid of our tool "ec2ldap" in Java and rewrote Cerveza in Python using Boto[29] (Amazon API binding for Python). We rewrote Cerveza to handle full instance start/stop/reboot with profile management. We chose Python over Java because of the scripting nature of Python. We didn't want to slow ourselves down in a compile/release process for this simple tool. A scripting language lets us add features quickly and do quick bug fixing.

Our previous outage led us to stop using SQLite. We wanted a solution where we do not have to rely on a local database or to be forced to start/stop instances from a management server. We replaced SQLite for Amazon SimpleDB[30] to store only profile information. For the rest we leverage the Tagging feature of the Amazon API. All our hosts or EBS volumes are tagged with hostname, device name, etc. This gives us much more flexibility as we can run Cerveza from our own laptop. We are not depending on the location of our SQLite database, we can start, stop, reboot instances from anywhere for any kind of server we want to start. The other major thing we got rid is the home made AMI. It takes lot of time to build and maintain an AMI, so it's not practical to deploy changes, etc. We chose to move to the official Ubuntu EC2 AMI and use cloud-init[31]. This is powerful. Cloud-init allow us to kick off our instance with different profiles by passing advanced user-data or scripts.

When starting a host with Cerveza for the first time we need to specify the instance profile we want to start (Hadoop node, MySQL, Java server, etc):

cerveza -m noc -- --zone ap-southeast-1a --start demo01 --profile UbuntuGeneric32Bit

To stop the host:

cerveza -m noc -- --zone ap-southeast-1a --stop demo01

To start the host a second time, we don't need to define the profile again, cerveza know it by querying SimpleDB :

cerveza -m noc -- --zone ap-southeast-1a --start demo01

Besides using LDAP for DNS data and SimpleDB for profiles information of existing hosts, Cerveza also uses Yaml[32] to define our instances profiles and volume profiles.

```
--- !InstanceProfile
name: UbuntuGeneric32Bit
desc: Ubuntu Generic instance profile without EBS
Volumes
aws: !InstanceAws
   ami: { us-east-1: ami-a6f504cf, us-west-1:
ami-957e2ed0, ap-southeast-1: ami-7c423c2e, ap-
northeast-1: ami-3a0fa43b, eu-west-1: ami-339ca947 }
   security_group: devzone
   key_pair: tm-devzone
   type: c1.medium
   elastic_ip: false
volumes: [ ]
startup_scripts: [ ]
shutdown_scripts: [ shutdown ]
user_data: [ cloud-config-base.txt, setup-hostname.sh,
root-login.sh, cloud-config-puppet.txt ]
check_ec2_kernel: 2.6.35-28-virtual
```

Our Ubuntu Generic 32 Bit instance is generally used for development purpose. In this profile we just define some basic information (instance type, key pair, default SG, AMI, etc.) but also important user-data. By passing a list of files, Cerveza will automatically concat all the given file to generate a compressed mime-multipart data file and pass it in the user-data when launching the instance. Cloud-init will read it and execute each script when the server boot. Cloud-init allow advanced configuration and many possibilities. In our case, the user-data script cloud-config-puppet.txt let us configure Puppet[33], our configuration management tool, at boot time.

B. Use a configuration management tool

We were thinking about using a configuration management tool for a long time, but hesitated until LISA 10. As we changed our AMI and started to use cloud-init, we took the opportunity to deploy puppet on all our hosts and start using it. We briefly looked at Cfengine[34] and Chef[35] too, but finally decided to go with Puppet as it seemed a little more documented and already fully integrated to Cloud-init.

Configuring and deploying puppet is fast and easy but using it properly is not that obvious. We had to deal with a couple of annoying problems like huge CPU spikes on each client, obscure errors for non-initiate people, process not running because of a lock file after reboot, etc. We addressed most of those issues. We found out that abusing of Augeas[36] is not necessarily good. We were able to speed up our puppet run from over 400 seconds to less than 15 seconds by replacing Augeas by puppet templates (mostly on long sysctl configuration). We use some ruby environment variables[37] to optimize each puppet client run, though we are still experimenting those. We stopped running puppet as a daemon as "fileserver" used too much resources. We had cases were puppet was using over 1GB of ram leading OOM Killer to kill some other process like our Membase[38] server. We now setup our puppet in a crontab running every half an

hour. To avoid a peak of requests on our puppet master we run the cron at random minutes on each client.

```
  # schedule puppet to run via cron
  $minute1 = generate('/usr/bin/env', 'sh', '-c',  'printf $((RANDOM
%29+0))')
  cron {
    "puppet_run":
      ensure      => present,
      command     => "/usr/sbin/puppetd --onetime --no-daemonize --
logdest syslog > /dev/null 2>&1",
      environment => [ 'RUBY_HEAP_MIN_SLOTS=500000',
                       'RUBY_HEAP_SLOTS_INCREMENT=250000',
                       'RUBY_HEAP_SLOTS_GROWTH_FACTOR=1',
                       'RUBY_GC_MALLOC_LIMIT=500000'
                     ],
      user        => "root",
      minute      => $minute1,
      hour        => "*";
  }
```

In the end, Puppet makes our life easier to manage and change configuration on multiple servers in four different data centers. Our puppet masters are located in our Colo center on US east coast. They are setup with Apache 2 + Phusion Passenger[39] with one master and one failover server. The failover server also handles the puppet reports using Puppet Dashboard[40]. We patched the puppet clients to report their FQDN as hostname instead of using there certificate name.

We currently don't have a clear dev environment for our puppet configuration, though our dev servers are setup to use a different environment so we can test our modules changes in dev before pushing to production. We are looking at better ways to manage this.

in puppet.pp:

```
class puppet inherits puppet::init {
  if $hostname =~ /^dev-*$/ or $ec2_security_groups ==
"devzone" {
     augeas {
       "puppet_env":
         context => "/files/etc/puppet/puppet.conf/main",
         onlyif  => "get environment != 'development'",
         changes => "set environment 'development'",
         notify  => Exec["puppet"];
     }
  }
}
```

in puppet.conf:

```
[development]
  manifestdir    =   $confdir/dev/manifests
  manifest       =   $manifestdir/site.pp
  modulepath     =   $confdir/dev/modules:$confdir/modules
```

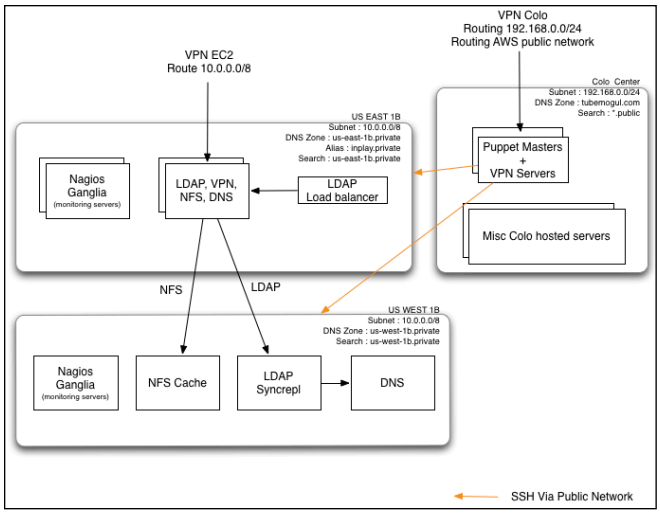

Fig 5. Network Flow between multiple AWS regions

C. Mirroring DNS, LDAP, NFS

Because of the multi-region and our response time constraint, we had to get DNS servers on each region. We use some "gateway" servers whose role is to serve as local DNS server, LDAP and NFS. As our DNS depend on LDAP, we initially setup LDAP Proxy with query caching which was working great except when running a non-cached query. We were getting some latency spike of up to four seconds for a DNS response. This was affecting our production response time in some cases increasing our percentage of timed out requests. We changed this configuration to use LDAP syncrepl[41]. Each LDAP server on each region is a master replicating one of our master server on US EAST. This solved our DNS response time and pam ldap response time. Though, since we use Autofs for our home directories we had to address the problem for our NFS server. On each region we use a NFSv4 mount with FS-Cache (cachefilesd[42]), this aimed to improve read speed on each region. The key thing we did was to remove the NFS mount point from the updatedb configuration because it would generally kill the server performance.

/etc/updatedb.conf:

```
PRUNE_BIND_MOUNTS="yes"
PRUNEPATHS="/tmp /var/spool /media /opt/openldap/var /
EBS /home"
PRUNEFS="NFS nfs nfs4 rpc_pipefs afs binfmt_misc proc
smbfs autofs iso9660 ncpfs coda devpts ftpfs devfs mfs shfs sysfs
cifs lustre_lite tmpfs usbfs udf fuse.glusterfs fuse.sshfs ecryptfs
fusesmb devtmpfs bindfs"
```

We are still not fully satisfied of our current solution and may stop using NFS for our home directory as it introduces a possible snowball effect in case our NFS fails on US east. Auto-mounted home directory doesn't give us any more added value as the product matures and our server

infrastructure grows. Also, we are having more clients using the NFS doing multiple mount/unmount leading to frequent home directories being stuck with a "Stale NFS file handle"[43].

D. What else?

To speed up our application deployment in multiple regions we started to use Amazon S3[44] with localized buckets. Instead of pushing our files from our Colo to each server, we push the files once to each of the localized S3 buckets then fetch the files to release on S3 from each server and deploy them locally.

Overall, with this infrastructure, we still have room for many improvement:

1) One clear blocker is NFS, we definitely plan to entirely remove NFS with auto-mounted home directory and get back to a more standard way to manage our servers. We are introducing more security checks and rules limiting production access so there shouldn't be any more need of user home directory being synchronized this way on all our servers.
2) We currently have two different sets of VPN and LDAP servers, one in our Colo and one in EC2. We want to centralize them to simplify our user and ACLs management.
3) We still have some "Gateway" servers, doing bridge between our regions. They are not based on the Ubuntu EC2 AMI. For lower maintenance on our side, we want to migrate everything onto the official Ubuntu EC2 AMI and fully use Cloud-init possibilities. We also want to get to a more standardized approach of managing our setup by using our internal Debian repository when required.
4) We are looking at Amazon VPC[45] to be able to better manage our private IPs and clusters. It can help to have better security policies in place preventing your backend from being accessed int the public internet, etc.
5) We plan to look again at Amazon ELB[46] to manage our different load balancing. One of the biggest drawbacks we had with ELB was the lake of visibility. No access logs and no clear error reporting make things hard to troubleshoot especially when you start having 500 errors returned by ELB during traffic spike.

V. Lesson learned

Evolution of your infrastructure must stay fault-tolerant in any case. What was simple and working at first can get complex in a multi-region / high latency environments.

In a small team with limited resources you will have little time to get everything right. You will miss important point leading to outages. Make sure to have a valid backup strategy and have a recovery procedure.

Never build a SPOF, even if it's for a "non-critical" use. As you start to rely more on this services (and you generally don't see it coming), your SPOF can have more impact than you would anticipate.

Infrastructure legacy can become a pain to maintain. Don't be afraid to revisit what you did and change it. What was true at one point of your design may not be true anymore.

Scaling your infrastructure in a fast paced environment require a lot of automation. which is why using a configuration management tool early would prevent you many headaches later on.

Acknowledgments

I would like to thank the LISA Chair and my shepherd, Marc Staveley, for the opportunity of this paper. It's an insightful experience that I would not hesitate to recommend to anyone.

I also want to thanks my close friends and family who continuously support me in my career choices.

This paper wouldn't have been possible without the opportunity I got by moving to the USA and joining TubeMogul in 2008 after just few Skype interviews. Hence, I express all my respect and consideration to John Hughes and Brett Wilson, TubeMogul's Founders.

References

[1] **Amazon Web Service (AWS)**
Amazon Web Services (AWS) delivers a set of services that together form a reliable, scalable, and inexpensive computing platform "in the cloud".
Website: http://aws.amazon.com

[2] **Amazon Elastic Cloud (EC2)**
Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers.
Website: http://aws.amazon.com/ec2

[3] **Amazon Instance**
An Amazon Instance is the AWS version of a server. It's known to be a Xen DomU Virtual Machine. Instances come in a variety of configurations and are designed to provide predictable and dedicated computing power on demand.

[4] **Availability Zone (AZ) and Regions**
Amazon EC2 provides the ability to place instances in multiple locations. Amazon EC2 locations are composed of Availability Zones and Regions. Regions are dispersed and located in separate geographic areas (US, EU, etc.). Availability Zones are distinct locations within a Region that are engineered to be isolated from failures in other Availability Zones and provide inexpensive, low latency network connectivity to other Availability Zones in the same Region.

[5] **Matching EC2 Availability Zone Across AWS Account**
By Eric Hammond on July 28, 2009
"*Summary*: EC2 availability zone names in different accounts do not match to the same underlying physical infrastructure. This article explains a trick which can be used to figure out how to match availability zone names between different accounts."
Blog post: http://alestic.com/2009/07/ec2-availability-zones

[6] **Amazon Machine Image (AMI)**
An Amazon Machine Image (AMI) is an encrypted machine image stored in Amazon S3. It contains all the information necessary to boot instances of your software.

[7] **Amazon Instance Type**
A specification that defines the memory, CPU, storage capacity, and hourly cost for an instance. Some instance types are designed for standard applications while others are designed for CPU-intensive applications.
Link: http://aws.amazon.com/ec2/instance-types

[8] **Amazon Security Group (SG)**
A security group is a named collection of access rules. These access rules specify which ingress (i.e., incoming) network traffic should be delivered to your instance. All other ingress traffic will be discarded.

[9] **Amazon Elastic IP (EIP)**
Elastic IP addresses are static IP addresses designed for dynamic cloud computing. An Elastic IP address is associated with your AWS account not a particular instance, and you control that address until you choose to explicitly release it. Unlike traditional static IP addresses, however, Elastic IP addresses allow you to mask instance or Availability Zone failures by programmatically remapping your public IP addresses to any instance in your account.

[10] **Amazon Elastic Block Store (EBS)**
Amazon Elastic Block Store (EBS) provides block level storage volumes for use with Amazon EC2 instances. Amazon EBS volumes are off-instance storage that persists independently from the life of an instance. Amazon Elastic Block Store provides highly available, highly reliable storage volumes that can be attached to a running Amazon EC2 instance and exposed as a device within the instance.
Website: http://aws.amazon.com/ebs

[11] **OpenVPN** "is a free and open source software application that implements virtual private network (VPN) techniques for creating secure point-to-point or site-to-site connections in routed or bridged configurations and remote access facilities. It uses SSL/TLS security for encryption and is capable of traversing network address translators (NATs) and firewalls." in Wikipedia: The Free Encyclopedia.
Website: http://openvpn.net

[12] **Auth-LDAP** plugin for OpenVPN
Website: http://code.google.com/p/openvpn-auth-ldap

[13] **OpenLDAP** is an open source implementation of the Lightweight Directory Access Protocol.
Website: http://www.openldap.org

[14] **BIND** is by far the most widely used DNS software on the Internet. It provides a robust and stable platform on top of which organizations can build distributed computing systems with the knowledge that those systems are fully compliant with published DNS standards.
Website: http://www.isc.org/software/bind

[15] Our Bind 9 install is patched with bind9-ldap + internal patch to support our LDAP schemas and specifics EC2 needs.
Website: http://bind9-ldap.bayour.com

[16] **Typica** is Java client library for a variety of Amazon Web Services.
Website: http://code.google.com/p/typica

[17] **SQLite** is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite is the most widely deployed SQL database engine in the world. The source code for SQLite is in the public domain.
Website: http://www.sqlite.org

[18] **Single Point Of Failure (SPOF)**
"A single point of failure (SPOF) is a part of a system that, if it fails, will stop the entire system from working.[1] They are undesirable in any system with a goal of high availability, be it a network, software application or other industrial system. Systems are made robust by adding redundancy in all potential SPOF and is generally achieved in computing through high-availability clusters. Redundancy can be achieved at the internal component level, at the system level (multiple machines), or site level (replication)." in Wikipedia: The Free Encyclopedia.

[19] **Ganglia** "is a scalable distributed system monitor tool for high-performance computing systems such as clusters and grids. It allows the user to remotely view live or historical statistics (such as CPU load averages or network utilization) for all machines that are being monitored." in Wikipedia: The Free Encyclopedia.
Website: http://ganglia.info

[20] **Nagios** "is a popular open source computer system and network monitoring software application. It watches hosts and services, alerting users when things go wrong and again when they get better." in Wikipedia: The Free Encyclopedia.
Website: http://www.nagios.org

[21] **Munin** is a networked resource monitoring tool that can help analyze resource trends and "what just happened to kill our performance?" problems. It is designed to be very plug and play. A default installation provides a lot of graphs with almost no work.
Website: http://munin-monitoring.org

[22] **Graphing Java JMX Object values with Ganglia and Python using JPype**
Blog post: http://goo.gl/LL7X3

[23] **Java Management Extensions (JMX)**
Set of specifications for application and network management in the J2EE development and application environment

[24] **JPype** is an effort to allow python programs full access to java class libraries. This is achieved not through re-implementing Python, as Jython/JPython has done, but rather through interfacing at the native level in both Virtual Machines.
Website: http://jpype.sourceforge.net

[25] **Nagios Service Check Acceptor (NSCA)**
NSCA allows you to integrate passive alerts and checks from remote machines and applications with Nagios. Useful for processing security alerts, as well as deploying redundant and distributed Nagios setups.
Website: http://goo.gl/ikagM

[26] **RRDtool** is the OpenSource industry standard, high performance data logging and graphing system for time series data. RRDtool can be easily integrated in shell scripts, perl, python, ruby, lua or tcl applications.
Website: http://oss.oetiker.ch/rrdtool

[27] **rrdcached** is a daemon that receives updates to existing RRD files, accumulates them and, if enough have been received or a defined time has passed, writes the updates to the RRD file.
Website: http://oss.oetiker.ch/rrdtool/doc/rrdcached.en.html

[28] **Haproxy** is a "Reliable, High Performance TCP/HTTP Load Balancer"
Website: http://haproxy.1wt.eu

[29] **Boto** is a Python interface to Amazon Web Services
Website: http://code.google.com/p/boto

[30] **Amazon SimpleDB (SDB)**
Amazon SimpleDB is a highly available, flexible, and scalable non-relational data store that offloads the work of database administration. Developers simply store and query data items via web services requests, and Amazon SimpleDB does the rest.
Website: http://aws.amazon.com/simpledb

[31] **Cloud-init** is the Ubuntu package that handles early initialization of a cloud instance. It is installed in the UEC Images and also in the official Ubuntu images available on EC2.
Website: https://help.ubuntu.com/community/CloudInit

[32] **YAML** is a human friendly data serialization standard for all programming languages.
Website: http://yaml.org

[33] **Puppet** is an open source configuration management tool.
Website: http://puppetlabs.com

[34] **CFEngine** automates IT processes and ensures the availability and consistency of applications and services.
Website: http://cfengine.com

[35] **Chef** is an open-source systems integration framework built specifically for automating the cloud. No matter how complex the realities of your business, Chef makes it easy to deploy servers and scale applications throughout your entire infrastructure. Because it combines the fundamental elements of configuration management and service oriented architectures with the full power of Ruby, Chef makes it easy to create an elegant, fully automated infrastructure.
Website: http://www.opscode.com/chef

[36] **Augeas** is a configuration editing tool. It parses configuration files in their native formats and transforms them into a tree. Configuration changes are made by manipulating this tree and saving it back into native config files.
Website: http://augeas.net

[37] **Fine tuning your garbage collector** By Chris Heald on June 13, 2009
Blog post: http://goo.gl/5GYBL

[38] **Membase** Server is the lowest latency, highest throughput NoSQL database technology on the market. When your application needs data, right now, it will get it, right now. A distributed key-value data store, Membase Server is designed and optimized for the data management needs of interactive web applications, so it allows the data layer to scale out just like the web application logic tier – simply by adding more commodity servers.
Website: http://www.couchbase.org/membase

[39] **Phusion Passenger**, aka mod_rails or mod_rack, allow easy and robust deployment of Ruby on Rails application on Apache and Nginx Webservers.
Website: http://www.modrails.com

[40] **Puppet Dashboard** is a web interface and reporting tool for your Puppet installation. Dashboard facilitates management and configuration tasks, provides a quick visual snapshot of important system information, and delivers valuable reports. In the future, it will also serve to integrate with other IT tools commonly used alongside Puppet.
Website: http://puppetlabs.com/puppet/related-projects/dashboard

[41] The **LDAP Sync Replication engine**, syncrepl for short, is a consumer-side replication engine that enables the consumer LDAP server to maintain a shadow copy of a DIT fragment. A syncrepl engine resides at the consumer and executes as one of the slapd(8) threads. It creates and maintains a consumer replica by connecting to the replication provider to perform the initial DIT content load followed either by periodic content polling or by timely updates upon content changes.
Documentation: http://www.openldap.org/doc/admin24/replication.html

[42] The **cachefilesd** daemon manages the cache data store that is used by network filesystems such a AFS and NFS to cache data locally on disk.
Man page: http://linux.die.net/man/8/cachefilesd

[43] **Stale NFS file handle**
Note: http://sysunconfig.net/unixtips/stale_nfs.txt

[44] **Amazon Simple Storage Service (S3)**
Amazon S3 provides a simple web services interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any developer access to the same highly scalable, reliable, secure, fast, inexpensive infrastructure that Amazon uses to run its own global network of web sites. The service aims to maximize benefits of scale and to pass those benefits on to developers.
Website: http://aws.amazon.com/s3

[45] **Amazon Virtual Private Cloud (VPC)**
Amazon Virtual Private Cloud (Amazon VPC) lets you provision a private, isolated section of the Amazon Web Services (AWS) Cloud where you can launch AWS resources in a virtual network that you define. With Amazon VPC, you can define a virtual network topology that closely resembles a traditional network that you might operate in your own datacenter. You have complete control over your virtual networking environment, including selection of your own IP address range, creation of subnets, and configuration of route tables and network gateways.
Website: http://aws.amazon.com/vpc

[46] **Amazon Elastic Load Balancing (ELB)**
Elastic Load Balancing automatically distributes incoming application traffic across multiple Amazon EC2 instances. It enables you to achieve even greater fault tolerance in your applications, seamlessly providing the amount of load balancing capacity needed in response to incoming application traffic. Elastic Load Balancing detects unhealthy instances within a pool and automatically reroutes traffic to healthy instances until the unhealthy instances have been restored. You can enable Elastic Load Balancing within a single Availability Zone or across multiple zones for even more consistent application performance.
Website: http://aws.amazon.com/elasticloadbalancing