



Auto scaling HTCondor Schedd using Amazon Web Services

November 1st 2014

Version 0.9

Claudio Pontili, Gabriele Garzoglio, Steven Timm

1	HTCondor and Schedd.....	3
2	Issues	3
2.1	Custom metrics.....	3
2.2	Scaling down policy	3
2.3	Scaling up policy	4
3	Publishing custom metric to AWS Cloudwatch.....	4
3.1	Pricing	4
4	Autoscaling group lifecycle.....	5
4.1	State Diagram of EC2 VM inside ASG	5
4.2	Lifecycle hooks.....	6
5	HTCondor Autoscaling architecture	7
5.1	Scaling Up vs Scaling Down	7
6	Tests	11
6.1	SSL certificate and Cname DNS record	11
6.2	Scaling up and down.....	11
7	Integration between AWS Condor-Schedd and FermiCloud Condor-Schedd.....	12
7.1	AWS Route 53.....	12
7.2	Integration architecture	12

7.3	Delegating a subdomain to AWS Route 53	13
7.3.1	Problems delegating fnal.gov subdomain	14
7.4	Testing the solution	14

1 HTCONDOR AND SCHEDD

HTCondor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, HTCondor provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to HTCondor, HTCondor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.

Like any other software solution, each HTCondor server has limit of traffic that can handle. In these pages we'll show you a way to use Amazon Web Service to install, configure, scale up and down HTCondor. Finally we'll see a possible integration between HTCondor inside FermiCloud with AWS to handle spikes of traffic.

2 ISSUES

The architecture of HTCondor is designed to scale up easily and you can split the traffic between two or more servers.

But if we want to use the commercial cloud like Amazon Web Service we'd like to scale up and down dynamically the number of servers because we pay for what we use.

Doing that means solve some problems:

2.1 CUSTOM METRICS

We'd like to scale up and down our HTCondor architecture trying to follow the number of jobs submitted to the servers.

AWS permits to create and terminate instances based on thresholds of metrics. AWS provides only few basic metrics about our EC2 Instances like CPU, IO, Network In and Out, etc. so they are not enough for our architecture.

The solution is to create a custom metric and put information from HTCondor server inside EC2 Instances to AWS Cloudwatch

<http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/publishingMetrics.html>

2.2 SCALING DOWN POLICY

With a custom AWS CloudWatch Metric we can set up a threshold to choose when our architecture must scale down. So using Autoscaling group and Custom Metrics we can terminate instances automatically when we don't need them.

But the termination process is not so easy, we'd like:

- To choose what instance we'd like to terminate (E.g. the instance with less idle jobs)
- To wait at least 5 days before terminating the instance. In fact we must give to the user the time to retrieve the output of his job.

To solve these problems we must handle the lifecycle of our EC2 instances inside the Auto Scaling Group.

2.3 SCALING UP POLICY

The scaling up policy should be the easiest part. Just use the custom metric and a threshold to create a new instance using an AMI (Amazon Machine Image) when we need it.

But we're using a pay-per-use commercial cloud and we'd like to minimize the cost. Therefore, before creating a new instance we'll try to find a terminating instance waiting 5 days to give the user the time to retrieve the output of his job.

Also in this case the solution is to manage the lifecycle of our EC2 instances inside the Auto Scaling Group

3 PUBLISHING CUSTOM METRIC TO AWS CLOUDWATCH

To scale up and down our architecture properly the best option is to create some custom metric putting information from inside our HTCondor servers to AWS CloudWatch.

The easiest way for doing that is to create a bash script using AWS CLI. The script will extract two information using `condor_status` statements:

- # of running jobs
- # of idle jobs

For instance the statement to put information inside CloudWatch is:

```
aws cloudwatch put-metric-data --metric-name TotalRunningJobs --namespace "TestCondor" --value $trunningj --region us-west-2
```

We can aggregate the custom metric using instance id or autoscaling group name, the information will be kept for 15 days.

3.1 PRICING

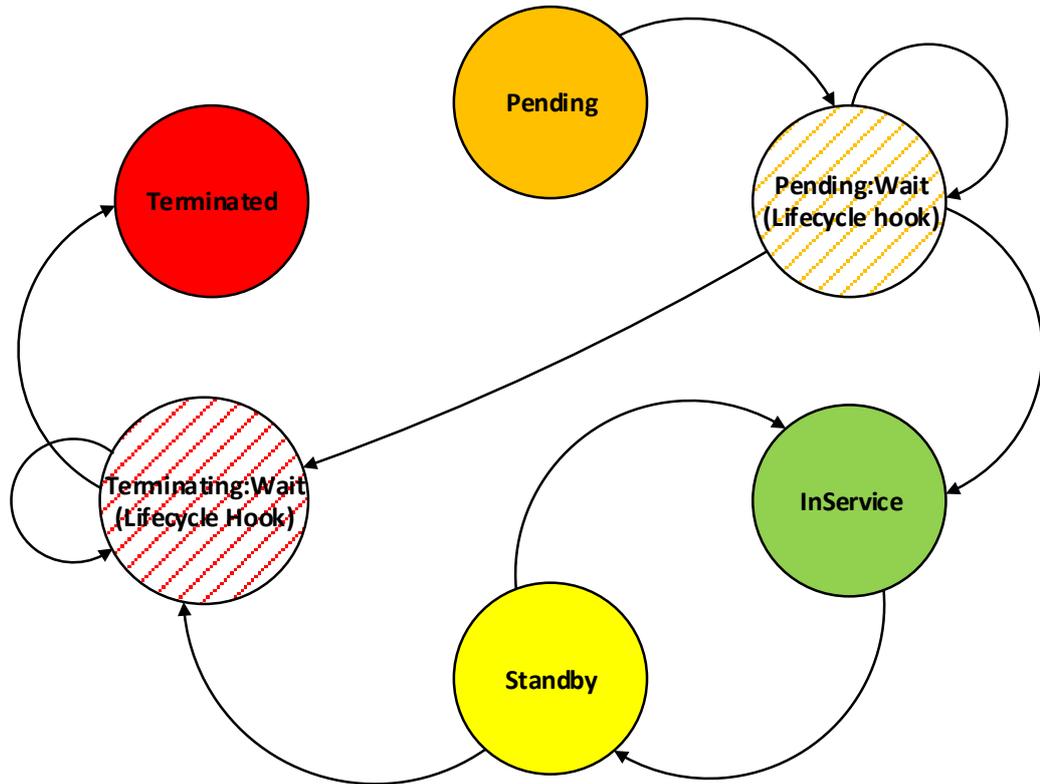
The price of this service is very low, in three weeks of testing we spent only few dollars

- \$0.01 per 1,000 GetMetricStatistics, ListMetrics, or PutMetricData requests
- \$0.50 per metric per month
- \$0.10 per alarm per month

4 AUTOSCALING GROUP LIFECYCLE

Each EC2 instance inside an AutoScaling Group has a state. Using lifecycle hooks, we can expand the number of these states and execute custom code when we need.

4.1 STATE DIAGRAM OF EC2 VM INSIDE ASG



When the ASG starts a new instance, it enters the *pending* state. The Operating System is starting.

Using lifecycle hook the instance can enter in the *pending:wait* state. The virtual machine is notified to enter in the new state using AWS SNS (Simple Notification Service). During this state, we can run custom code and choose if we want to terminate the instance or move it to InService.

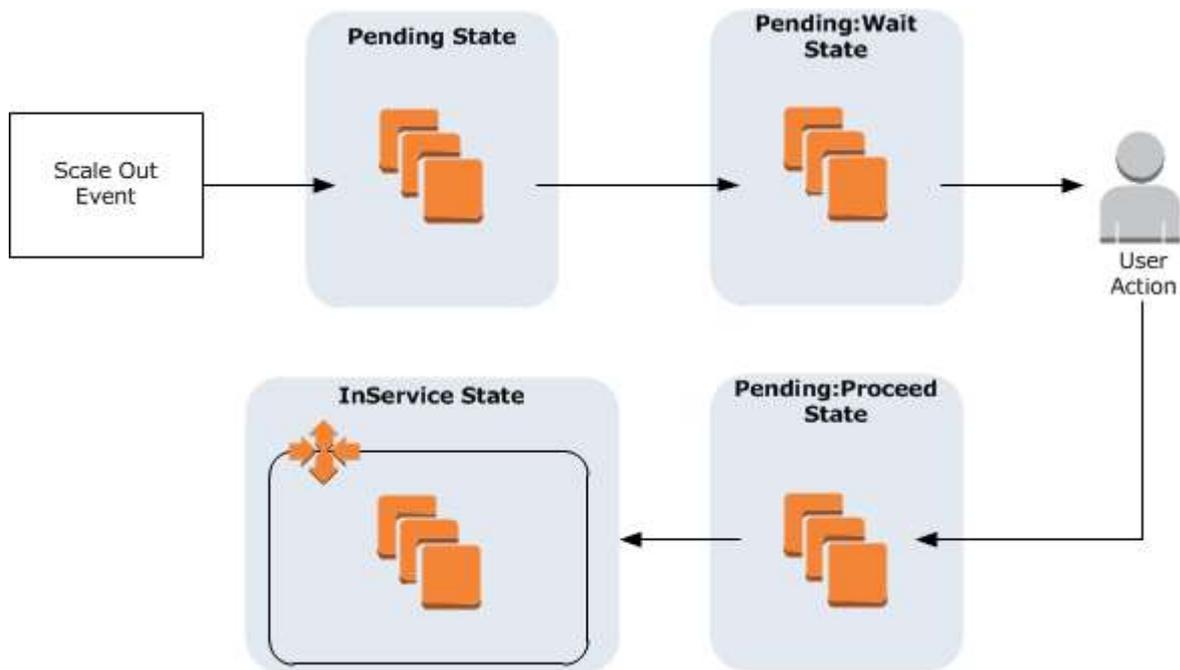
When the instance is *InService* it's up and running. Moreover it's registered to the ELB (Elastic Load Balancer) and can receive jobs from users.

We can move an instance to *Standby* using AWS API. In this state the instance is up and running and can be reached with the public ip address but it's de-registered from the ELB and it doesn't receive job from users. The instance can be either terminated or come back to Inservice

Using lifecycle hooks the instance can enter in *Terminating:wait* state. The virtual machine is notified to enter in this state using AWS SNS (Simple Notification Service). During this state, we can run custom code and the virtual machine is reachable using the public ip address. A terminating:wait instance cannot be restored to InService.

When an instance is *terminated* the data is deleted and it cannot be restored.

4.2 LIFECYCLE HOOKS

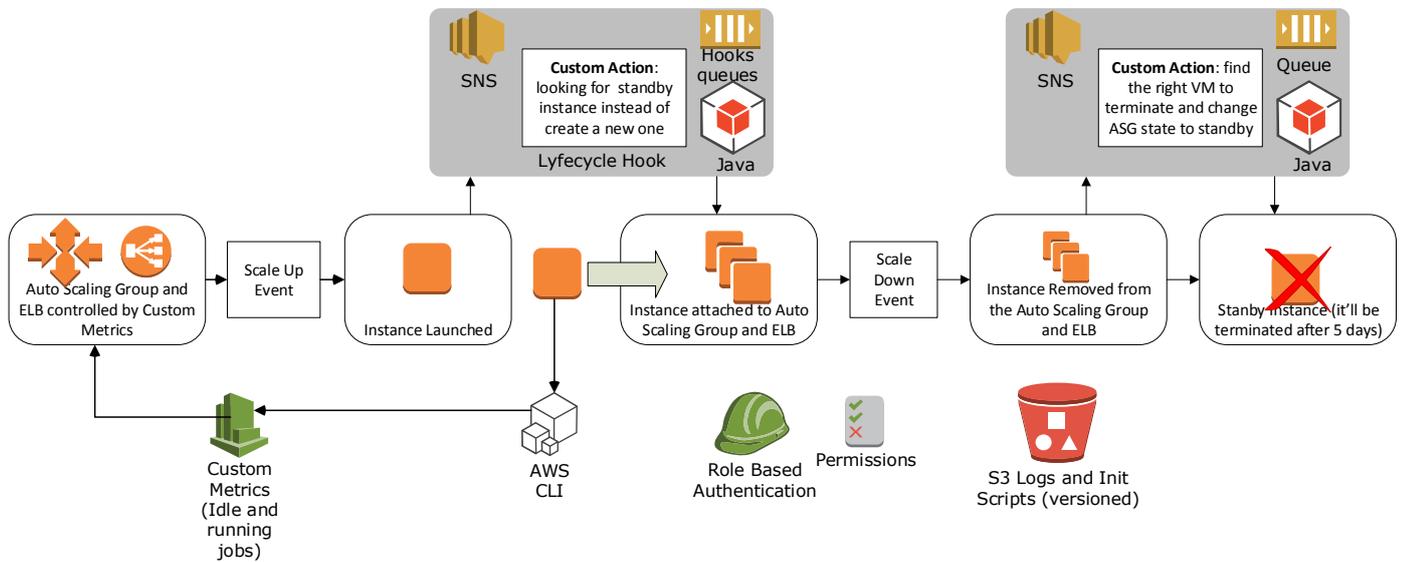


You can configure a set of Lifecycle actions for each of your Auto Scaling Groups. Messages will be sent to a notification target for the group (an [SQS](#) queue or an [SNS](#) topic) each time an instance enters the *Pending* or *Terminating* state. Your application is responsible for handling the messages and implementing the appropriate initialization or decommissioning operations.

After the message is sent, the instance will be in the *Pending:Wait* or *Terminating:Wait* state, as appropriate. Once the instance enters this state, your application is given 60 minutes to do the work. If the work is going to take more than 60 minutes, your application can extend the time by issuing a "heartbeat" to Auto Scaling. If the time (original or extended) expires, the instance will come out of the wait state.

After the instance has been prepared or decommissioned, your application must tell Auto Scaling that the lifecycle action is complete, and that it can move forward. This will set the state of the instance to *Pending:Proceed* or *Terminating:Proceed*.

5 HTCONDOR AUTOSCALING ARCHITECTURE

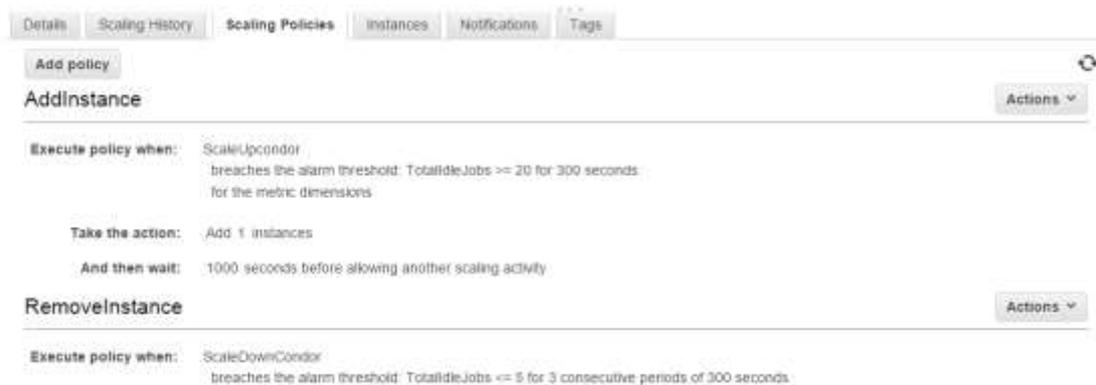


This is a diagram of the architecture created and tested inside AWS. The list of the components is:

- An AutoScalingGroup that control when and how to scale up and down. The ASG contains a Launch configuration where we can set the AMI, user-data (to install the additional software and script inside the HTCondor at runtime), the instance type (e.g. m3.medium), security group, etc.
- An Elastic load balancer where the instances “InService” are registered. Only instances registered to ELB can receive new jobs.
- At least two custom metrics. The custom metrics are created using AWS CLI
- A pending:wait lifecycle hook to execute custom code that looks for a standby instance instead of creating a new one during the scale up event. The instance is notified to enter in the *pending:wait* state using Simple Notification Service (SNS) and Simple Queue Service (SQS) using AWS Java SDK.
- A CloudWatch Alarm. The alarm send a message to SNS, from SNS the message is sent to another SQS. After reading the message from SQS, an instance is chosen and moved to Standby.
- Either AWS CLI or AWS Java SDK use Role Based Authentication. Using it an access key and secret key is automatically generated and rotated based on the Role attached to the EC2 instance. The role is configured inside the Launch Configuration.
- S3 store the logs of the ELB and the Init Scripts of the EC2 instances. Instead of installing all the software using EC2 Userdata inside the Launch Configuration, a number of script are download from S3 and executed.

5.1 SCALING UP VS SCALING DOWN

Scaling policies are configured inside the ASG using AWS Web Console



But the process to scale up and down is different. When we scale up we use lifecycle hooks *pending:wait* but when we scale down we use *standby* state. We cannot use lifecycle hooks *terminating:wait* because an instance entering in this state cannot come back to “InService”

When we scale up:

- The instance enter *pending:wait* state using lifecycle hook
- The lifecycle hook notifies the new state to the instance using Simple Notification Service (SNS)
- SNS send a message to Simple Queue Service (SQS)
- Using AWS Java SDK the new instance read the message from SQS and extract a lifecycle action token from the message
- Using AWS Java SDK the new instance check if there is a *standby* instance inside the ASG
- If there is *standby* instance inside the ASG, the new instance is terminated using the lifecycle action token. Moreover the *standby* instance is moved to *Inservice*
- If there is no *standby* instance inside the ASG, the new instance is moved to *Inservice* using the lifecycle action token.

When we scale down:

- A CloudWatch alarm is configured to send a message to SNS when the architecture reaches the scale down threshold.
- The message is sent to SQS. The configuration of this queue can guarantee that the message can be read only one time.
- Each HTCondor server *Inservice* check the SQS every 5 minutes. When one of the server read the message, it try to find the server with the lowest number of idle jobs and move it to “standby” using AWS Java SDK.
- Moreover, another script is installed inside each HTCondor Server and it runs every 5 minutes. The script checks if the instance is *Standby*, there is no running or idle jobs and the last completed job is at least 5 days ago. In this case, the instance is terminated. The script uses AWS CLI and the statements `condor_status` and `condor_history`

5.1.1 SQS Configuration

To scale up and down we use queues. To have the right behaviour we had to configure two parameters:

- Default Visibility Timeout: the length of time that a message received from a queue will be invisible to other receiving components
- Message retention period: the amount of time that Amazon SQS will retain a message if it doesn't get deleted

The configuration of the SQS for the pending:wait lifecycle hook was:

Configure LifecycleHooksUP Cancel

Queue Settings

Default Visibility Timeout:	<input type="text" value="5"/>	<input type="text" value="seconds"/>	Value must be between 0 seconds and 12 hours.
Message Retention Period:	<input type="text" value="1"/>	<input type="text" value="hours"/>	Value must be between 1 minute and 14 days.
Maximum Message Size:	<input type="text" value="256"/>	KB	Value must be between 1 and 256 KB.
Delivery Delay:	<input type="text" value="0"/>	<input type="text" value="seconds"/>	Value must be between 0 seconds and 15 minutes.
Receive Message Wait Time:	<input type="text" value="0"/>	seconds	Value must be between 0 and 20 seconds.

Dead Letter Queue Settings

Use Redrive Policy:	<input type="checkbox"/>		
Dead Letter Queue:	<input type="text"/>		Value must be an existing queue name.
Maximum Receives:	<input type="text"/>		Value must be between 1 and 1000.

We used these parameters because we can create more than 1 instance at the same time and in this case the second instance could read the token of the first. So inside the java code we checked this problem and we retry after 5 seconds.

The configuration of the SQS for the Cloudwatch allarm to scale down is:

Configure CondorScaleDown Cancel

Queue Settings

Default Visibility Timeout:	<input type="text" value="1"/>	hours ▼	Value must be between 0 seconds and 12 hours.
Message Retention Period:	<input type="text" value="1"/>	hours ▼	Value must be between 1 minute and 14 days.
Maximum Message Size:	<input type="text" value="256"/>	KB	Value must be between 1 and 256 KB.
Delivery Delay:	<input type="text" value="0"/>	seconds ▼	Value must be between 0 seconds and 15 minutes.
Receive Message Wait Time:	<input type="text" value="0"/>	seconds	Value must be between 0 and 20 seconds.

Dead Letter Queue Settings

Use Redrive Policy:	<input type="checkbox"/>	
Dead Letter Queue:	<input type="text"/>	Value must be an existing queue name.
Maximum Receives:	<input type="text"/>	Value must be between 1 and 1000.

As you can see setting the default visibility timeout at the same time of the message retention period we want to minimize the possibility that the message is read by 2 instances.

We can just minimize because SQS is a distributed system. In fact reading the SQS FAQ

“Yes, under rare circumstances you might receive a previously deleted message again. This can occur in the rare situation in which a DeleteMessage operation doesn’t delete all copies of a message because one of the servers in the distributed Amazon SQS system isn’t available at the time of the deletion. That message copy can then be delivered again. You should design your application so that no errors or inconsistencies occur if you receive a deleted message again.”

But this is not a big problem. First, it’s very rare. Second, if you instance read the same message at the same instant they will run the same code and probably select the same server to move to standby. If they select two different server after few minutes, the ASG will trigger a scale up process and one of these instance come back in service.

6 TESTS

To test the architecture we've used m3.medium EC2 instance type and an ASG with minimum of 1 virtual machine and maximum 2.

The instance AMI is ami-e72863d7

6.1 SSL CERTIFICATE AND CNAME DNS RECORD

For testing purpose we've created a SSL certificate for the subdomain fermicondor.vexpert.it

Moreover with a DNS CNAME Record the subdomain fermicondor.vexpert.it has been redirected to the DNS name of the ELB Schedd-1212275796.us-west-2.elb.amazonaws.com.

6.2 SCALING UP AND DOWN



In the blue colour, we can see the average number of idle jobs inside our HTCondor server inside AWS.

So after reaching the scaling up threshold, another Condor Server is created automatically and the average number idle jobs drop down because the new server is empty.

We continued to submit jobs. During this amount of time the ELB divided the jobs between two servers.

After the job computation is completed the average number of idle jobs start to fall down slowly.

When we reached the scaling down threshold, an instance is selected and moved to standby state. At the same time the instance is deregistered from the ELB.

7 INTEGRATION BETWEEN AWS CONDOR-SCHEDD AND FERMICLOUD CONDOR-SCHEDD

A way to integrate the AWS and the Fermi solution of Condor-Schedd is to use appropriate DNS record to redirect the traffic automatically to commercial cloud when it is up and running.

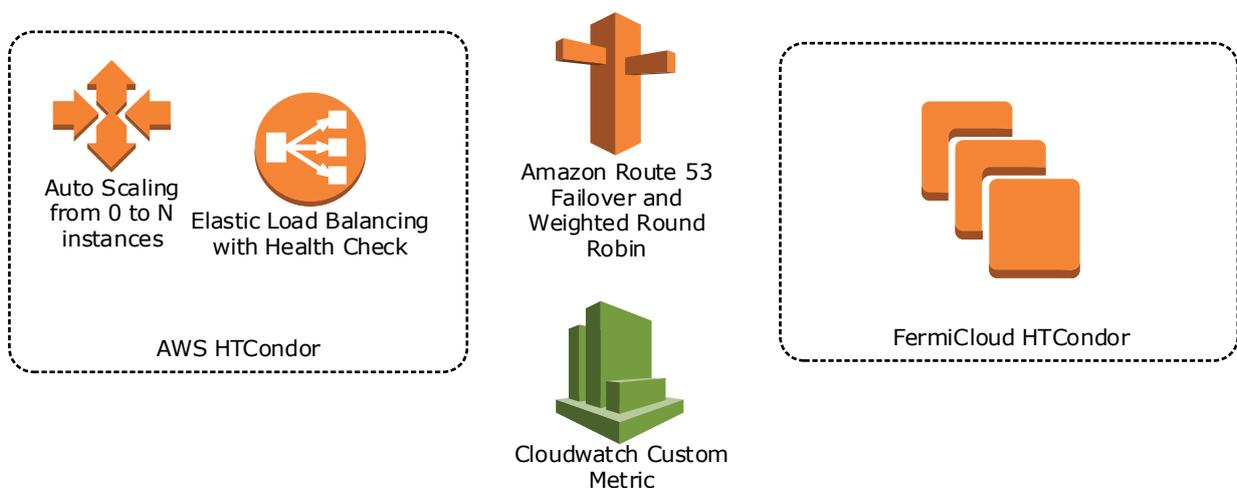
7.1 AWS ROUTE 53

Amazon Route 53 is a highly available and scalable cloud Domain Name System (DNS) web service. It is designed to give developers and businesses an extremely reliable and cost effective way to route end users to Internet applications by translating names like `www.example.com` into the numeric IP addresses like `192.0.2.1` that computers use to connect to each other.

Amazon Route 53 effectively connects user requests to infrastructure running in AWS – such as Amazon EC2 instances, Elastic Load Balancing load balancers, or Amazon S3 buckets – and can also be used to route users to infrastructure outside of AWS. You can use Amazon Route 53 to configure DNS health checks to route traffic to healthy endpoints or to independently monitor the health of your application and its endpoints. Amazon Route 53 makes it possible for you to manage traffic globally through a variety of routing types, including Latency Based Routing, Geo DNS, and Weighted Round Robin—all of which can be combined with DNS Failover in order to enable a variety of low-latency, fault-tolerant architectures. Amazon Route 53 also offers Domain Name Registration – you can purchase and manage domain names such as `example.com` and Amazon Route 53 will automatically configure DNS settings for your domains.

Moreover you can update all the record and configurations of Route 53 using AWS CLI or SDKs.

7.2 INTEGRATION ARCHITECTURE



We can integrate AWS HTCondor and FermiCloud HTCondor using AWS to handle spikes of traffic.

During tests we had:

- An ASG with a minimum of zero instances
- An ELB with a Health check that controls when the instances are up and running testing TCP port 8443
- A custom metric. Both the AWS HTCondor and FermiCloud HTCondor put statistics inside the metric
- A DNS Failover record inside Route 53 attached to the health check of the ELB. So when there is at least one instance inside AWS the traffic is redirected to the public cloud. When there is no instance the traffic is redirected to FermiCloud.
- A DNS Weighted Round Robin record inside Route 53 to redirect only a fraction of the traffic to AWS (E.g. 1/3 of traffic to AWS and 2/3 of traffic to FermiCloud)

7.3 DELEGATING A SUBDOMAIN TO AWS ROUTE 53

Instead of using Route 53 as DNS for an entire domain, we can delegate only one subdomain. In this way, we can manage only a couple of subdomain (E.G. condorintegration.fnal.gov or fifebatch-dev.fnal.gov).

For doing that we have to create 4 NS records (we need 4 to have HA) inside the main DNS Server as it's explained here:

<http://docs.aws.amazon.com/Route53/latest/DeveloperGuide/CreatingNewSubdomain.html#UpdateDNSParentDomain>

NS (Nameserver) ⓘ				
10 Records (0 Selected)				
✓	Host	Points To	TTL	Actions
<input type="checkbox"/>	@ (Informational)	ns55.domaincontrol.com (Informational)	1 Hour (Informational)	
<input type="checkbox"/>	@ (Informational)	ns56.domaincontrol.com (Informational)	1 Hour (Informational)	
<input type="checkbox"/>	condoraws	ns-409.awsdns-51.com	1/2 Hour	 
<input type="checkbox"/>	condoraws	ns-675.awsdns-20.net	1/2 Hour	 
<input type="checkbox"/>	condoraws	ns-1378.awsdns-44.org	1/2 Hour	 
<input type="checkbox"/>	condoraws	ns-1662.awsdns-15.co.uk	1/2 Hour	 
<input type="checkbox"/>	condorintegration	ns-409.awsdns-51.com	1/2 Hour	 
<input type="checkbox"/>	condorintegration	ns-675.awsdns-20.net	1/2 Hour	 
<input type="checkbox"/>	condorintegration	ns-1378.awsdns-44.org	1/2 Hour	 
<input type="checkbox"/>	condorintegration	ns-1662.awsdns-15.co.uk	1/2 Hour	 

In the picture you can see the configuration of two subdomain: condoraws.publiccloudexpert.com and condorintegration.publiccloudexpert.com hosted in GoDaddy.

7.3.1 Problems delegating fnal.gov subdomain

We're not sure that we could delegate a subdomain of fnal.gov so we can advise a possible solution.

Instead of using Route 53 we can create a load balancer inside FermiCloud. The load balancer has to check the AWS API Autoscaling to understand if the Amazon architecture is up and running. In this case it redirect the traffic to Amazon, in the other case inside Fermicloud.

Of course this solution is more complex and needs code to call AWS API.

7.4 TESTING THE SOLUTION

For instance, we've delegated two subdomain of domain publiccloudexpert.com hosted on Godaddy:

- condoraws.publiccloudexpert.com
- condorintegration.publiccloudexpert.com

We have attached the first one to health check of the ELB to control if there at least one running instance inside our AWS Condor-Schedd architecture. If there's, the dns record reply with the CNAME of the ELB otherwise it sends the traffic to the ip address 131.225.155.133 (your current dev-condor-schedd inside fermicloud).

So when the traffic is low we can turn off our AWS infrastructure and the traffic is automatically redirected to fermi cloud.

The second record is a weighted Round Robin that sends 1/3 of the traffic to condoraws.publiccloudexpert.com and 2/3 of the traffic to fifebatch-dev.fnal.gov.

Route53 is a powerful service because it can be updated using REST API (or SDKs) and for example we can update the weight of the RR in function of the current number of servers inside AWS.

Last but not least, what happens if there's a fail of the connectivity of Fermilab and it remains isolated? If you use the weighted RR DNS, 1/3 of the traffic will fail and to solve the problem we'll need to update the records of the fermilab DNS servers manually. Not the best thing to do of course, this's the only drawback.