Nube híbrida: Autoescalado horizontal hacia AWS

HashiCorp

Terraform

Josué Álvarez Moreno

Índice

Prefacio.

Todos los scripts de provisionamiento y de transición de local-a-cloud están disponibles en https://github.com/j-alvarez-moreno/proyecto-2ASIR bajo licencia GPL 3.0 para cualquier uso que quieras darle. Este documento incluye sólo el código de los más relevantes conceptualmente.

Este repositorio no es un programa *plug and play* que puedas aplicar a tu infraestructura y ya tienes autoescalado horizontal, pero puede servirte de punto de partida para adaptarlo a tus necesidades.

Dicho esto, comencemos.

Escenario tipo sobre el cual trabajaremos.

Estado inicial.

Nuestra empresa hipotética, **JosuéOnSecurity,** tiene un blog en el que escribimos artículos semi-técnicos con el objetivo de captar clientes. Hasta ahora, este blog ha sido servido desde nuestro centro de datos. Conforme hemos ido recibiendo más visitantes únicos a lo largo del tiempo, la estrategia de escalado ha consistido *simplemente* en añadir más RAM y CPU a la máquina. No disponemos de *failover* de ningún tipo, ya que el resultado de un análisis coste/beneficio muestra que se requiere gran inversión inicial y un retorno económico difícilmente cuantificable: nuestro blog puede generar buenas sensaciones sobre nuestra efectividad como proveedor de soluciones de seguridad, pero la *buena imagen* no aparece en la lista de generadores de tráfico hacia la sección de ventas.

Contexto semi-ficticio.

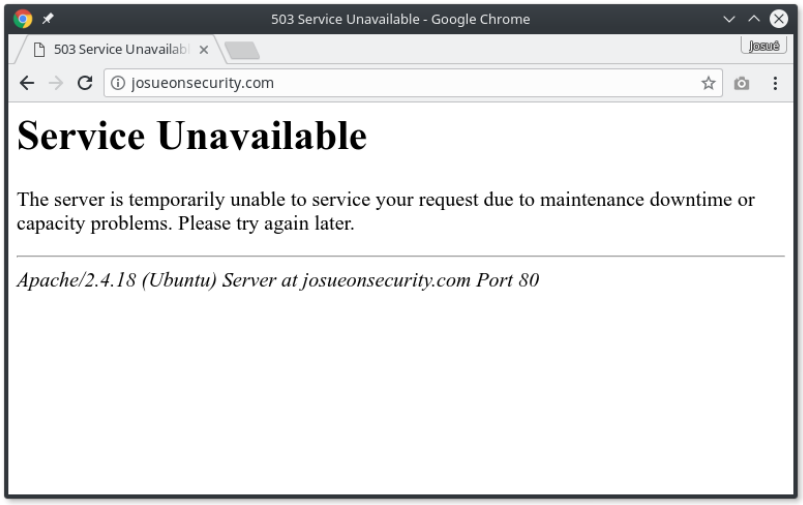
El 12 de Mayo de 2017 se inició un ataque de *ransomware* a gran escala que paralizó a instituciones como Telefónica, el equivalente británico a la sanidad española (NHS), FedEx y el homónimo alemán de Renfe (Deutsche Bahn). Explicado brevemente, el cryptogusano conocido como WannaCry cifra los datos del disco duro y solicita un rescate abonado a través de BitCoin.

JosueOnSecurity dispone de un producto que protege contra las infecciones de ransomware, así que rápidamente lanzamos una campaña de PR sobre lo fácil que hubiese sido para estas grandes organizaciones evitar esta catástrofe. Esta campaña consiste en varias entradas en el blog cuidadosamente creadas para incrementar nuestro SEO y contacto personalizado con clientes que tenemos en el punto de mira.

El problema.

¡Éxito! Nuestros posts empiezan a recibir cientos de visitas de usuarios únicos por segundo. Nos llaman de El País, quieren sacarnos en un árticulo como *expertos* y lo quieren ya. Google Analytics nos indica que alguien ha posteado nuestro artículo en Hacker News, Reddit y Xataka. Está ganando tracción a gran velocidad, y algunos usuarios de nuestro software hablan de lo bien que les ha funcionado y que de como ellos don't wannacry.

Nuestro CEO desciende al subsótano de IT con una botella de whiskey de casi cuatro cifras para felicitarnos personalmente por cómo hemos podido aprovechar esta oportunidad gracias a nuestra maravilloso equipo de IT y, mientras nos cuenta todas las enormes posibilidades que esta atención nos ha otorgado…



...oh.

Objetivo de este documento.

¿Cómo podemos planear capacidad para picos de popularidad imprevisibles, sin desperdiciar dinero y hardware el resto del año? Simple y sencillo, aplicamos la nube a nuestra arquitectura de forma que podemos **escalar de forma horizontal bajo demanda** cuando sea necesario y reducir de nuevo nuestro *hardware* al volumen habitual tras el pico de demanda.

- Debe ser un proceso **automatizado** que no requiera de intervención manual: Nuestra disponibilidad tiene que tener tantos nueves como sea posible.
- Debe ser **fiable**: Funcionará hoy, mañana y el año que viene independientemente del contenido de la web.
- Debe ser **flexible**: Queremos poder añadir más poder de computación o reducirlo en tiempo real, ajustándonos a la demanda.

Solución.

- Monitorizar el uso de recursos, e iniciar el proceso de escalado hacia la nube de Amazon cuando se supere un criterio definido. Para esto usaremos **Nagios**.
- El despliegue de la infraestructura en la nube debe ser 100% automatizado, y debe ser suficientemente flexible para reflejar nuestra arquitectura actual *y futura* en la nube. De esta forma, no tendremos que tener en cuenta donde se van a desplegar los recursos y podremos considerarlo **un segundo centro de datos desplegable bajo demanda en breves minutos**. Usaremos **Terraform**.
    - Añadir y eliminar nodos en la nube debe ser un proceso totalmente automatizado, incluyendo su registro en Nagios y Haproxy.
- Servir recursos web es un tipo de problema *fácilmente* paralelizable: Cada usuario único que visite nuestra web tiene su propio contexto, y puede residir en su propio “nodo” sin compartir estado con otros usuarios. Para balancear la carga entre estos nodos usaremos **Haproxy**.
    - Un caso no contemplado en esta implementación es el uso de base de datos. Este apartado lo estudiaremos de forma teórica en el anexo dedicado a mejoras de esta solución.

Ventajas.

- **Infraestructura como código** (IaC): Podremos *versionar/revertir/expandir* nuestra infraestructura de forma no destructiva.
- Inversión inicial requerida: **Despreciable**, podemos hacer las pruebas en instancias t2.micro y pagaremos unos 2 dólares al día por levantar varios nodos.

- El tiempo de despliegue es literalmente cinco minutos **en total**. Terraform genera nuestra infraestructura de forma paralela, así que no importa si estamos añadiendo dos nodos o cuarenta.
- Terraform es **agnóstico**: Al contrario que CloudFormation o Heat, Terraform soporte diferentes proveedores otorgándonos la flexibilidad de usar el que más nos convenga en cada momento.

Inconvenientes y detalles a considerar.

- General
    - No existe una herramienta *plug and play* para esta situación, sino múltiples componentes de software que podemos unir programáticamente de forma que obtenemos el resultado deseado. Gran parte de nuestro trabajo es escribir scripts muy adaptados a nuestro entorno específico.
        - Dicho esto, hay varios puntos lógicos en el diseño para inyectar componentes adicionales como bases de datos en HA o aplicaciones.
- Terraform
    - Su fiabilidad depende enteramente de la API de cada proveedor: Aunque la herramienta intenta mitigar posibles fallos mediante reintentos, timeouts y el uso de una caché local que contiene el estado de la infraestructura, está a merced de la fiabilidad de la API. Con AWS me he encontrado con situaciones en las que Terraform es incapaz de generar o destruir una instancia porque la *id* no existe, a pesar de que dicha *id* aparece en la consola de AWS asociada a la instancia deseada.
        - AWS es conocida por la *fiabilidad* de su dashboard de salud de la API: Tick verde significa todo correcto, tick verde con una pequeña interrogación significa caída generalizada.
    - La caché local ("*terraform.state*") es el punto de referencia autoritario y absoluto para Terraform. Si se corrompe o la nube se modifica desde fuera de Terraform, podemos encontrarnos en una situación en la que Terraform *destruya y regenere* recursos de forma innecesaria y en ocasiones causando caídas.
    - La flexibilidad de proveedores tiene un *precio*: **Aunque Terraform es agnóstico, la sintáxis declarativa de los ficheros no lo es**. Por tanto, debemos mantener diferentes *versiones* de nuestra infraestructura según dónde vayamos a desplegarla.

A continuación explicaremos el uso de Terraform, Nagios y Haproxy; describiendo la configuración que utilizaremos para implementar esta solución y cómo adaptaremos cada uno de estos componentes para que encajen en el puzzle final.

Detalles preliminares.

Todas las máquinas contienen un usuario puppet, sin contraseña, que permitirá conexiones con nuestra clave privada. Este usuario tendrá privilegios de *sudo* sin requerir contraseña, y será nuestro punto de entrada para automatizar las tareas.
Esta parte del provisionamiento está recogida en `base-provisioning.sh`.

El control del dns del dominio lo realizaremos a través de `ddclient`, tanto en el centro de datos local como en el remoto.

Dado que mi entorno de pruebas local está basado en Vagrant, durante el provisionamiento almaceno los recursos en /vagrant/ para poder usar los mismos scripts y fingir que Vagrant es mi *centro de datos* con ajustes mínimos.

Nagios.

Nagios es una aplicación open-source que monitoriza sistemas y redes, y es capaz de enviar alertas y ejecutar acciones cuando cambia la disponibilidad de los componentes monitorizados. Vamos a implementarlo utilizando un servidor central, y desplegaremos el agente de Nagios en cada uno de los nodos para monitorizar el uso de CPU. Cuando el uso supere el nivel que consideremos adecuado, utilizaremos Nagios para iniciar las tareas de escalado hacia la nube.

Aunque aquí vamos a detallar el proceso paso a paso, esto estará automatizado en el escenario final mediante `nagios-server.sh`.

Instalación y configuración del servidor.

```
sudo DEBIAN_FRONTEND=noninteractive apt-get -y install nagios-plugins
nagios3 nagios-nrpe-plugin
```

Para poder forzar chequeos sobre los servicios y hosts de forma manual, debemos cambiar el valor de la directiva `check_external_commands` a 1 en el archivo `/etc/nagios3/nagios.cfg`.

También debemos cambiar el usuario y modo de dos directorios para permitir la ejecución de estos, ya que Nagios ejecuta estos chequeos a través de un archivo temporal que debe poder ser ejecutable por el usuario del servidor web. Dado que el archivo se recrea en cada ejecución, vamos a fijar los permisos a nivel del directorio.

```
# dpkg-statoverride --update --add nagios www-data 2710
/var/lib/nagios3/rw
```

```
# sudo dpkg-statoverride --update --add nagios nagios 751
/var/lib/nagios3
```

Para habilitar el acceso web, creamos un usuario llamado ‘nagiosadmin’ con contraseña ‘nagios’ y reiniciamos el servicio:

```
# htpasswd -b -c /etc/nagios3/htpasswd.users nagiosadmin nagios
# service nagios3 restart
```

Dado que queremos que Nagios detecte subidas de carga de CPU lo más rápido posible, vamos a cambiar el intervalo *base* de 60 segundos a 1 segundo. Este intervalo es multiplicado por el factor especificado en el fichero que define al host a monitorizar.

```
# sudo sed -i 's/interval_length=.*/interval_length=1/g'
/etc/nagios3/nagios.cfg
```

Configuración de notificaciones por email.

Para que Nagios nos envíe un email cuando ocurra algún evento, necesitaremos:

- Un programa para el envío de correo instalado.
- Credenciales de nuestro proveedor de email.
- Definir el comando adecuado en commands.cfg.

En mi caso, voy a utilizar `sendemail` para el envío de correo a través de mi cuenta de GMail. Lo instalamos desde los repositorios:

```
# apt-get install sendemail
```

Añadimos las definiciones de las variables $USER4 y $USER5 que corresponden a nuestro usuario y contraseña de GMail, en el archivo `/etc/nagios3/resource.cfg`:
$USER4$=usuario
$USER5$=contraseña

Y definimos los comandos adecuados para la notificación de cambio de estado en `/etc/nagios3/commands.cfg`

```
# 'notify-host-by-email' command definition
define command{
       command_name  notify-host-by-email
        command_line /usr/bin/printf "%b" "***** Nagios *****nnNotification Type:
$NOTIFICATIONTYPE$nHost: $HOSTNAME$nState: $HOSTSTATE$nAddress: $HOSTADDRESS$nInfo:
$HOSTOUTPUT$nnDate/Time: $LONGDATETIME$n" | /usr/bin/sendEmail -xu $USER4$ -xp $USER5$
-t $CONTACTEMAIL$ -f $CONTACTEMAIL$ -o tls=auto -s smtp.gmail.com  -u "**
$NOTIFICATIONTYPE$ Host Alert: $HOSTNAME$ is $HOSTSTATE$ **" -m "***** Nagios
```

```
*****nnNotification Type: $NOTIFICATIONTYPE$nHost: $HOSTNAME$nState:
$HOSTSTATE$nAddress: $HOSTADDRESS$nInfo: $HOSTOUTPUT$nnDate/Time: $LONGDATETIME$n"
}

# 'notify-service-by-email' command definition
define command{
       command_name  notify-service-by-email
        command_line /usr/bin/printf "%b" "***** Nagios *****nnNotification Type:
$NOTIFICATIONTYPE$nnService: $SERVICEDESC$nHost: $HOSTALIAS$nAddress:
$HOSTADDRESS$nState: $SERVICESTATE$nnDate/Time: $LONGDATETIME$nnAdditional
Info:nn$SERVICEOUTPUT$" | /usr/bin/sendEmail -s smtp.gmail.com -xu $USER4$ -xp $USER5$
-t $CONTACTEMAIL$ -f $CONTACTEMAIL$ -u "** $NOTIFICATIONTYPE$ Service Alert:
$HOSTALIAS$/$SERVICEDESC$ is $SERVICESTATE$ **" -m "***** Nagios *****nnNotification
Type: $NOTIFICATIONTYPE$nnService: $SERVICEDESC$nHost: $HOSTALIAS$nAddress:
$HOSTADDRESS$nState: $SERVICESTATE$nnDate/Time: $LONGDATETIME$nnAdditional
Info:nn$SERVICEOUTPUT$"
}
```

Reaccionar a eventos de carga de CPU en los nodos.

Definimos dos comandos: check_remote_load y su servicio asociado que obtendrá la carga de la CPU de los nodos y nagios-cpu-handler.sh, que iniciará el proceso de escalado a la nube cuando se genera una carga crítica de CPU. Los nodos pertenecerán todos a un grupo llamado *nodes* que tiene asociado el servicio de monitorización de carga de CPU.

```
sudo bash -c 'cat <<EOF > /etc/nagios3/conf.d/hostgroup-nodes.cfg

define hostgroup {
        hostgroup_name  nodes
                alias           Dynamic nodes
                members         *
        }
EOF'

sudo bash -c 'cat <<EOF >> /etc/nagios3/conf.d/services_nagios2.cfg
define service {
        hostgroup_name                  nodes
        service_description             Load
        event_handler                   nagios-cpu-handler
        check_command                   check_remote_load
        use                             generic-service
        notification_interval           0
}
EOF'

sudo bash -c 'cat <<EOF > /etc/nagios3/conf.d/resizing-commands.cfg
```

```
define command{

command_name nagios-cpu-handler
command_line /bin/nagios-cpu-handler.sh \$HOSTADDRESS$
\$SERVICESTATE$ \$SERVICESTATETYPE$ \$SERVICEATTEMPT\$
}
EOF'

sudo bash -c 'cat <<EOF >> /etc/nagios3/commands.cfg

define command {
    command_name    check_remote_load
    command_line \$USER1\$/check_nrpe -H \$HOSTADDRESS\$ -c
check_load
}

EOF'
```

nagios-cpu-handler.sh

Cuando ocurre algún evento, nagios llama al *event_handler* especificado en la definición del servicio monitorizado con los siguientes argumentos:

<IP HOST> <ESTADO SERVICIO> <CRITICIDAD> <NUMERO DE INTENTO>

Por ejemplo, sería el equivalente a ejecutar lo siguiente:

```
nagios-cpu-handler.sh 192.168.1.125 CRITICAL SOFT 3
```

Aunque sólo vayamos a reaccionar cuando el nivel sea crítico, tenemos que implementar todos los casos. Por tanto, el script quedaría de la siguiente forma:

```
#!/bin/sh
case "$2" in
OK)
     # No tenemos que hacer nada, el servicio está bien.
     echo "$1 IS OK" >> /tmp/service.state
     ;;
WARNING)
     # Ha subido la carga, pero aún no nos interesa hacer nada.
     ;;
UNKNOWN)
     # Error desconocido, no podemos tomar ninguna acción ya que no
sabemos lo que es.
```

```
      ;;
CRITICAL)
      case "$3" in
      SOFT)
            case "$4" in
            3)
                  # Vamos por el tercer reintento y la CPU sigue
estando cargada de forma crítica.
                  ;;
                  esac
            ;;

            # Cuarto intento, el equivalente a carga superior al 70%
durante más de un minuto. Dado que tenemos diferentes nodos que se
reparten la carga de forma más o menos equitativa, esto significa que
estan todos cargados.
            # Iniciamos el proceso de transición a la nube.
      HARD)
            /bin/init-to-the-cloud.sh
            ;;
      esac
      ;;
esac
exit 0
```

El script init-to-the-cloud.sh contiene la lógica necesaria para otorgar el control al servidor maestro de la nube, y pasa la batuta al mismo para que este inicie las configuraciones necesarias para una correcta transición.

```
#!/bin/bash
cloudMaster="34.204.91.248"


# We allow the cloud to kick in on proyecto.josuealvarezmoreno.es as
soon as it is ready.
sudo service ddclient stop

# Here we would call helper scripts to make sure the cloud and the
local-cache are in sync

# After we are fully ready, we hand over the baton to the cloud.

ssh -n -o BatchMode=yes -o ConnectTimeout=5 -o
StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null -i
```

```
/vagrant/provisioning-key admin@$cloudMaster "sudo bash -c
/bin/engage.sh > /tmp/engage.log"
```

Una vez en la nube, levantamos el número de nodos iniciales que consideremos adecuado según nuestra infraestructura, los añadimos al Haproxy y al Nagios principal, y actualizamos las DNS para que nuestro dominio ahora apunte a la nube.

Entraremos en detalle sobre estos scripts tras la sección de Terraform.

Plantilla de monitorización para los nodos.

Recargar el servicio de Nagios es suficiente para añadir cualquier configuración nueva almacenada en ./conf.d. Vamos a aprovecharlo para añadir ahí los nodos nuevos conforme los generemos, basándonos en la siguiente plantilla.

```
sudo bash -c 'cat <<EOF > /etc/nagios3/conf.d/nodes.template

define host{
        use                     generic-host
        host_name               #NODE-HOSTNAME#
        alias                   #NODE-HOSTNAME#
        address                 #NODE-IP#
        }

EOF'
```

Configuración de los nodos

Cada nodo deberá tener instalado el agente de monitorización, y definido el comando que usaremos para comprobar el uso de CPU. Adicionalmente, tenemos que habilitar la ejecución de comandos de forma remota y añadir al servidor de Nagios a la lista de hosts permitidos. Este proceso está implementado de forma programática en `nagios-client.sh`.

Instalación del agente de monitorización

```
DEBIAN_FRONTEND=noninteractive sudo apt-get install -y --force-yes
build-essential nagios-nrpe-server nagios-plugins
```

Habilitamos la ejecución de comandos de forma remota.

```
sudo sed -i "s/.*dont_blame_nrpe.*/dont_blame_nrpe = 1/"
/etc/nagios/nrpe.cfg
```

Definición del comando de chequeo de carga de CPU.

Vamos a utilizar el plugin check_load, que recibe los argumentos de la siguiente forma:

```
sudo bash -c 'echo
"command[check_load]=/usr/lib/nagios/plugins/check_load -w
0.5,0.4,0.3, -c 0.6,0.5,0.4" >> "/etc/nagios/nrpe_local.cfg"'
```

Estamos generando un warning cuando el uso de CPU sea de 0.5 en los últimos 5 minutos, 0.4 en los últimos 10 y 0.3 en los últimos 15. De la misma forma, generamos una alerta crítica si alcanzamos 0.6, 0.5 ó 0.4 respectivamente.

El uso de CPU se mide utilizando las facilidades del sistema, y estos valores están pensando para máquinas con un sólo núcleo. Si tuviesen dos, el valor de carga máxima sería 2.0; 3 núcleos sería 3.0… etc.

Añadir al servidor de Nagios a la lista de hosts permitidos.

```
sudo sed -i "s/.*allowed_hosts.*/allowed_hosts = 127.0.0.1
192.168.100.1/" /etc/nagios/nrpe.cfg
```

HAProxy.

Haproxy es un software open-source que proporciona Alta Disponibilidad y balanceo de carga para conexiones HTTP a través de TCP.

Lo implementaremos en el servidor maestro, y se encargará de repartir el tráfico a los diferentes nodos. Una propiedad importante de Haproxy es que puede aplicar nueva configuración sin denegar conexiones, por lo que podemos añadir y quitar nodos sabiendo que recargar la configuración no supone una interrupción del servicio.

Haproxy soporta muchos tipos de algoritmos para la distribución del tráfico, pero dos de los más importantes son **roundrobin** (el que usaremos) y **source**.

- Roundrobin reparte las peticiones de forma equitativa, garantizando que eventualmente todos los nodos recibirán el mismo número de peticiones.

- Source retiene cada usuario único en un nodo específico, y es útil si necesitamos mantener estado de una conexión a otra.

La configuración base de Haproxy es similar a la que hemos visto durante el curso, pero con dos diferencias:

- Ahora que vamos a tener un potencial gran número de nodos, deberíamos cambiar el número de conexiones soportadas a algo más razonable. Para este ejemplo, usaremos 20000.
- Para que los scripts automatizados inserten los nodos en el fichero de configuración, vamos a insertar dos anclajes de referencia al final del fichero /etc/haproxy/haproxy.cfg:

    #node_list_begin

    #node_list_end

Para añadir nodos utilizaremos un script que hemos llamado `haproxy-add-host.sh`. He añadido soporte para hostnames, aunque usaremos las ips como nombre de host. Al fin y al cabo son un rebaño, no mascotas.

```
#!/bin/bash
hostIP=$1
hostName=$2
configurationLine="     server $2 $1:8080 maxconn 300 check"
sudo sed -i "/#node_list_end/i\ $configurationLine"
/etc/haproxy/haproxy.cfg
sudo systemctl reload haproxy.service
```

Terraform.

Qué es

Terraform es software que nos permite definir nuestra infraestructura como código. Terraform procesa nuestro código, lo compara con el estado del proveedor de servicios especificado y construye un plan de ejecución para que el estado de la infraestructura desplegada sea el definido en el código.

Debido a esta forma de funcionar, podemos añadir nuevas instancias o modificar instancias y recursos en nuestro código (claves ssh, conectividad de red, reglas de firewall…) y aplicarlos a la infraestructura remota sin preocuparnos del *cómo*.

Esto nos permitirá añadir y eliminar nodos de computación de forma fácil y sencilla.

Instalación.

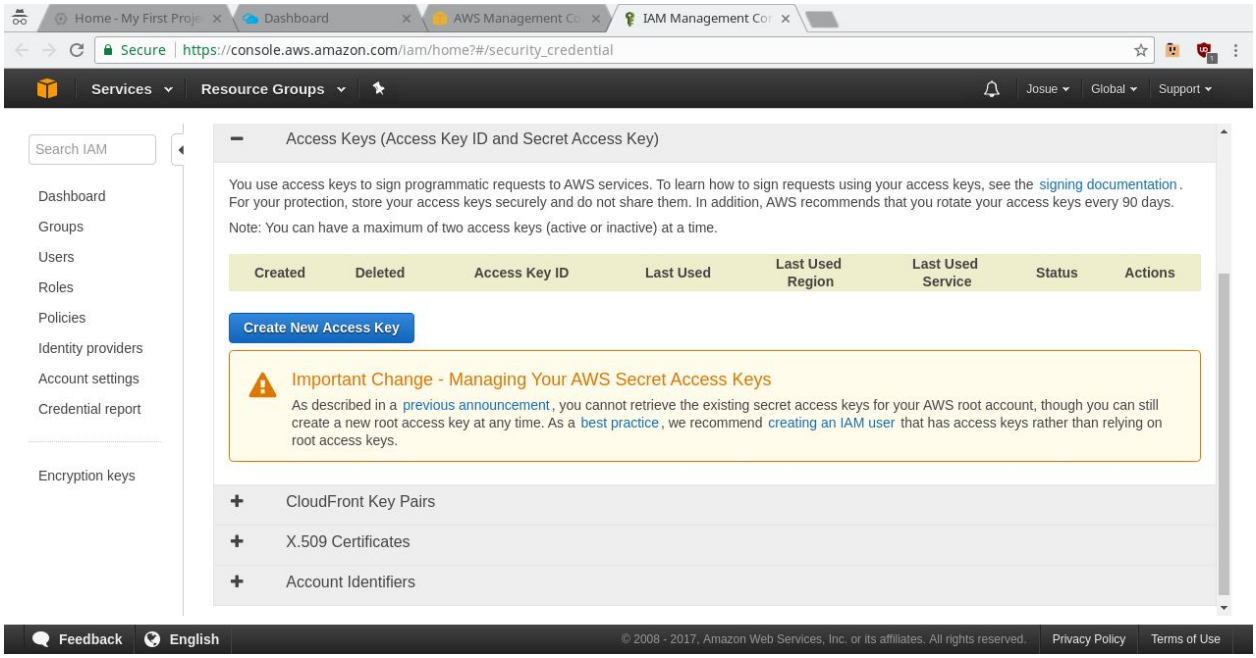
Descargamos la versión adecuada de https://www.terraform.io/downloads.html

```
sterling@Pegasus 13:51:59 /tmp wget
https://releases.hashicorp.com/terraform/0.9.5/terraform_0.9.5_linux_
amd64.zip
--2017-05-13 13:52:02--
https://releases.hashicorp.com/terraform/0.9.5/terraform_0.9.5_linux_
amd64.zip
Resolving releases.hashicorp.com (releases.hashicorp.com)...
151.101.1.183, 151.101.65.183, 151.101.129.183, ...
Connecting to releases.hashicorp.com
(releases.hashicorp.com)|151.101.1.183|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 30720943 (29M) [application/zip]
Saving to: 'terraform_0.9.5_linux_amd64.zip'

terraform_0.9.5_linux_amd64.zip
100%[=====================================================================
=====================================================================
=====>]  29,30M  16,2MB/s    in 1,8s

2017-05-13 13:52:04 (16,2 MB/s) - 'terraform_0.9.5_linux_amd64.zip'
saved [30720943/30720943]
```

```
sterling@Pegasus 13:52:04 /tmp unzip terraform_0.9.5_linux_amd64.zip
Archive:  terraform_0.9.5_linux_amd64.zip
  inflating: terraform
sterling@Pegasus 13:52:07 /tmp sudo cp terraform /usr/bin
```

Primeros pasos: Desplegar una instancia en AWS

Para que Terraform pueda acceder a los recursos del proveedor que hayamos elegido, necesitaremos claves API. En el caso de Amazon AWS, [podemos obtenerlas en la consola de AWS](#).



Una vez obtenidas las claves, vamos a desplegar una instancia de Ubuntu 16.04 en una máquina T2.micro. Generamos el fichero ejemplo.tf con el siguiente contenido:

```
#Definimos las claves de la API
provider "aws" {
  access_key = "ACCESS_KEY_HERE"
  secret_key = "SECRET_KEY_HERE"
  region     = "us-east-1"
}
# Detalles de la instancia
resource "aws_instance" "ejemplo" {
  ami           = "ami-2757f631"
  instance_type = "t2.micro"
  subnet_id     = "subnet-3fd11213"
}
# Clave pública SSH
resource "aws_key_pair" "ssh-keys" {
  key_name   = "terraform"
  public_key = "ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQDPjhXfdmdg0e5b82or4st2umGZ8Mng5QYuJJAHz
vjtW1+F9VObVKByAoA5YK6cscFTXL0as9agmKjdmepn4y36wiPB6EU7jZWEE3+nCbF+vp
sK9V8CP23N8q3p+sukz2FmKlT/pvt3+9JcinbxhYD9ASOhOjvO18Xx/PY8ErN2q+jjjJH
O9yT8aIZxRWD5znR0GLBv6s7li0d3xQNyoqiMNRXpEIi6a1pLn1z/rGdN1GM4bb3m0gl3
qvgZ4/XcGBmBqNtFjDc1pzJaksGhFgzE2yVqAIgLXKBm7OEAXTUAfaRU/rKZxmbHM7lG1
wMpkoVVPma/7wAq1ZKxluJV05ml sterling@Pegasus"
}
# Creamos una red virtual privada para al instancia
resource "aws_vpc" "default" {
  cidr_block = "10.0.0.0/16"
}

# Un gateway para que la máquina tenga acceso a internet
resource "aws_internet_gateway" "default" {
  vpc_id = "${aws_vpc.default.id}"
}
# Damos acceso a internet a la vpc
resource "aws_route" "internet_access" {
  route_table_id         = "${aws_vpc.default.main_route_table_id}"
  destination_cidr_block = "0.0.0.0/0"
  gateway_id             = "${aws_internet_gateway.default.id}"
}

# Una subnet para nuestra mñaquina
resource "aws_subnet" "default" {
```

```
  vpc_id                  = "${aws_vpc.default.id}"
  cidr_block              = "10.0.1.0/24"
  map_public_ip_on_launch = true
}

# Creamos un grupo de seguridad para poder abrir puertos en la vpc
resource "aws_security_group" "elb" {
  name        = "terraform_example_elb"
  description = "Grupo de seguridad de ejemplo"
  vpc_id      = "${aws_vpc.default.id}"

  # Abrimos el puerto 80 desde y hacia todas partes
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  # "-1" significa TCP y UDP
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
  # SUPER IMPORTANTE: Permitimos acceso SSH desde cualquier IP. Sin
esto, no podremos conectarnos tras levantar la máquina.
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
```

Terraform tiene cuatro comandos básicos: *refresh*, *plan, apply* y *destroy*. *Refresh* refresca la caché actual del estado de la infraestructura definida en el código, *plan* muestra los cambios que se realizarían comparando el estado del código con el del proveedor, y apply aplica los cambios. *Destroy* es evidente, y pide confirmación antes de borrar nada.

- **MUY IMPORTANTE: Apply no pide confirmación**. Cualquier diferencia entre el estado remoto y el definido localmente será solucionada *sin piedad.*
    - ¿Has cambiado la configuración del firewall para permitir el tráfico ICMP? ¡Listo!
    - ¿Has añadido otro par de claves SSH para autenticarte con el host bastión? ¡Aplicado!
    - ¿Estás haciendo pruebas y has movido el archivo *servidorOracle.tf* a otra carpeta y se te ha olvidado devolverlo a su sitio? **¡Purgando la instancia de Oracle y sus discos asociados, tal y como ha ordenado usted Capitán!**

Para protegernos de cambios hacia una instancia que **sabemos** no debe sufrir modificaciones, podemos añadir la directiva ignore_changes dentro de lifecyle. Podemos ignorar cambio en cualquier atributo de nuestro despliegue.

```
  lifecycle {
        ignore_changes = ["*"]
  }
```

Si inspeccionamos el plan generado por Terraform, observamos que va a crear la instancia especificada

```
$ terraform plan
+ aws_instance.example
    ami:                          "ami-2757f631"
    associate_public_ip_address:  "<computed>"
    availability_zone:            "<computed>"
    ebs_block_device.#:           "<computed>"
    ephemeral_block_device.#:     "<computed>"
    instance_state:               "<computed>"
    instance_type:                "t2.micro"
    ipv6_address_count:           "<computed>"
    ipv6_addresses.#:             "<computed>"
    key_name:                     "<computed>"
    network_interface.#:          "<computed>"
    network_interface_id:         "<computed>"
    placement_group:              "<computed>"
    primary_network_interface_id: "<computed>"
    private_dns:                  "<computed>"
```

```
    private_ip:                   "<computed>"
    public_dns:                   "<computed>"
    public_ip:                    "<computed>"
    root_block_device.#:          "<computed>"
    security_groups.#:            "<computed>"
    source_dest_check:            "true"
    subnet_id:                    "<computed>"
    tenancy:                      "<computed>"
    volume_tags.%:                "<computed>"
    vpc_security_group_ids.#:     "<computed>"
```

Plan: 1 to add, 0 to change, 0 to destroy.

Nota: este plan sólo existe en memoria. Al hacer *terraform apply* Terraform volverá a refrescar y el plan generado puede ser diferente si las circunstancias son diferentes.

- Es posible almacenar el plan con el switch *-out* y pasárselo a *apply*.

Nota 2: El error Error launching source instance: OptInRequired: You are not subscribed to this service ocurre porque el acceso por API no está habilitado. Para evitar un uso abusivo, Amazon tarda un tiempo indefinido entre 0 y 24 horas en habilitar *de verdad* las claves API.

```
$ terraform apply
aws_instance.example: Creating...
  ami:                               "" => "ami-2757f631"
  associate_public_ip_address:       "" => "<computed>"
  availability_zone:                 "" => "<computed>"
  ebs_block_device.#:                "" => "<computed>"
  ephemeral_block_device.#:          "" => "<computed>"
  instance_state:                    "" => "<computed>"
  instance_type:                     "" => "t2.micro"
  ipv6_address_count:                "" => "<computed>"
  ipv6_addresses.#:                  "" => "<computed>"
  key_name:                          "" => "<computed>"
  network_interface.#:               "" => "<computed>"
  network_interface_id:              "" => "<computed>"
  placement_group:                   "" => "<computed>"
  primary_network_interface_id:      "" => "<computed>"
  private_dns:                       "" => "<computed>"
  private_ip:                        "" => "<computed>"
  public_dns:                        "" => "<computed>"
  public_ip:                         "" => "<computed>"
  root_block_device.#:               "" => "<computed>"
  security_groups.#:                 "" => "<computed>"
```

```
  source_dest_check:            "" => "true"
  subnet_id:                    "" => "subnet-3fd11213"
  tenancy:                      "" => "<computed>"
  volume_tags.%:                "" => "<computed>"
  vpc_security_group_ids.#:     "" => "<computed>"
aws_instance.example: Still creating... (10s elapsed)
aws_instance.example: Still creating... (20s elapsed)
aws_instance.example: Creation complete (ID: i-02fbb95386573f5ec)

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Ahora podemos comprobar el estado (cacheado) de nuestra infraestructura con *terraform show*. Entre otros datos, tenemos la IP pública de la instancia.

```
aws_instance.example:
  id = i-02fbb95386573f5ec
  ami = ami-2757f631
  associate_public_ip_address = true
  availability_zone = us-east-1a
  disable_api_termination = false
  ebs_block_device.# = 0
  ebs_optimized = false
  ephemeral_block_device.# = 0
  iam_instance_profile =
  instance_state = running
  instance_type = t2.micro
  ipv6_address_count = 0
  ipv6_addresses.# = 0
  key_name =
  monitoring = false
  network_interface.# = 0
  network_interface_id = eni-0aaf6ac3
  primary_network_interface_id = eni-0aaf6ac3
  private_dns = ip-10-0-0-149.ec2.internal
  private_ip = 10.0.0.149
  public_dns = ec2-107-21-158-184.compute-1.amazonaws.com
  public_ip = 107.21.158.184
  root_block_device.# = 1
  root_block_device.0.delete_on_termination = true
  root_block_device.0.iops = 100
  root_block_device.0.volume_size = 8
  root_block_device.0.volume_type = gp2
  security_groups.# = 0
```

```
source_dest_check = true
subnet_id = subnet-3fd11213
tags.% = 0
tenancy = default
volume_tags.% = 0
vpc_security_group_ids.# = 1
vpc_security_group_ids.713989925 = sg-535d3d2d
```

Vamos a deshacernos de esta instancia de ejemplo con el comando *destroy*.

```
$ terraform destroy
Do you really want to destroy?
  Terraform will delete all your managed infrastructure.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes

aws_instance.example: Refreshing state... (ID: i-02fbb95386573f5ec)
aws_instance.example: Destroying... (ID: i-02fbb95386573f5ec)
aws_instance.example: Still destroying... (ID: i-02fbb95386573f5ec,
10s elapsed)
aws_instance.example: Still destroying... (ID: i-02fbb95386573f5ec,
20s elapsed)
aws_instance.example: Still destroying... (ID: i-02fbb95386573f5ec,
30s elapsed)
aws_instance.example: Still destroying... (ID: i-02fbb95386573f5ec,
41s elapsed)
aws_instance.example: Still destroying... (ID: i-02fbb95386573f5ec,
51s elapsed)
aws_instance.example: Still destroying... (ID: i-02fbb95386573f5ec,
1m1s elapsed)
aws_instance.example: Still destroying... (ID: i-02fbb95386573f5ec,
1m11s elapsed)
aws_instance.example: Destruction complete
```

Provisionamiento de instancias.

Tras crear la instancia, podemos copiar ficheros y directorios a la misma. Así mismo, con la directiva *remote-exec* podemos indicarle la ruta de un script *local* y Terraform lo copiará a la instancia y lo ejecutará.

Veámoslo con un pequeño script:
$ cat bootstrap.sh: echo 'hello' > /tmp/output.txt

Modificamos la definición de la instancia para añadir el provisionador, y la clave SSH que debe usar para conectarse.

```
resource "aws_instance" "instancia1" {
  ami           = "ami-b374d5a5"
  instance_type = "t2.micro"
  key_name = "terraform"
```

```
  provisioner "remote-exec" {
    scripts = [
      "bootstrap.sh"
    ]
    connection {
      type     = "ssh"
      user     = "ubuntu"
      private_key = "${file("~/.ssh/terraform-proyecto")}"
    }
  }
}
```

Tras aplicar el plan, observamos lo siguiente:

```
aws_instance.instancia1 (remote-exec): Connecting to remote host via
SSH...
aws_instance.instancia1 (remote-exec):   Host: 54.90.2.13
aws_instance.instancia1 (remote-exec):   User: ubuntu
aws_instance.instancia1 (remote-exec):   Password: false
aws_instance.instancia1 (remote-exec):   Private key: true
aws_instance.instancia1 (remote-exec):   SSH Agent: true
aws_instance.instancia1 (remote-exec): Connected!
aws_instance.instancia1: Creation complete (ID: i-03cbd14db6ba0a896)
```

Y efectivamente:

```
$ ssh -i ~/.ssh/terraform-proyecto ubuntu@54.234.144.181 "cat
/tmp//tmp/output.txt’”
Hello
```

Descripción del servidor maestro en la nube.

```
resource "aws_instance" "main" {
  ami           = "ami-b14ba7a7"
  availability_zone = "us-east-1a"
  instance_type = "t2.micro"
  key_name = "terraform"
# Dado que vamos a otorgar el control a la instancia de Terraform que ejecutaremos en la nube,
y esa instancia no tiene terraform.state… Terraform borraria el servidor maestro en el apply ya
que no coincide con lo que tiene cacheado. Para evitar esto, añadimos la siguiente directiva.
  lifecycle {
        ignore_changes = ["*"]
  }
# Un mejor diseño hubiese sido tener el servidor maestro siempre en nuestro centro de datos, y
hacer una VPN a la que se unan los nodos para monitorizar. Ver apéndice para más detalles.

  vpc_security_group_ids = ["${aws_security_group.default.id}"]

  subnet_id = "${aws_subnet.default.id}"

  provisioner "file" {
    source = "/home/sterling/Dropbox/media/clases/proyecto/provisioning-scripts"
    destination = "/tmp/vagrant"
  }

  provisioner "file" {
    source =
"/home/sterling/Dropbox/media/clases/proyecto/03-Terraform-provisioned-trial/credentials.tf"
    destination = "/tmp/vagrant/credentials.tf"
  }
  provisioner "file" {
    source =
"/home/sterling/Dropbox/media/clases/proyecto/03-Terraform-provisioned-trial/network-setup.tf"
    destination = "/tmp/vagrant/network-setup.tf"
  }
  provisioner "file" {
    source =
"/home/sterling/Dropbox/media/clases/proyecto/03-Terraform-provisioned-trial/master-instance.t
f"
    destination = "/tmp/vagrant/master-instance.tf"
  }
  provisioner "file" {
    source =
"/home/sterling/Dropbox/media/clases/proyecto/03-Terraform-provisioned-trial/variables.tf"
    destination = "/tmp/vagrant/variables.tf"
  }

  provisioner "remote-exec" {
    scripts = [
      "${var.provisioning-folder}/cloud-vagrant-glue.sh",
      "${var.provisioning-folder}/base-provisioning.sh",
      "${var.provisioning-folder}/control-provisioning.sh",
      "${var.provisioning-folder}/haproxy-provisioning.sh",
      "${var.provisioning-folder}/nagios-server.sh",
      "${var.provisioning-folder}/autoNodeScan-provisioning.sh",
      "${var.provisioning-folder}/ddclient-provisioning.sh",
```

```
      "${var.provisioning-folder}/terraform-autoresize-provisioner.sh"
    ]
  }


  connection {
    type     = "ssh"
    user     = "admin"
    private_key = "${file("${var.provisioning-folder}/provisioning-key")}"
  }
}
```

Plantilla que usaremos para los nodos.

Los nodos son generados por un script que realiza sustitución de variables en la siguiente plantilla.

```
resource "aws_instance" "#INSTANCE-NAME#" {
  ami           = "ami-b14ba7a7"
  availability_zone = "us-east-1a"
  instance_type = "t2.micro"
  key_name = "terraform"

  # Our Security group to allow HTTP and SSH access
  vpc_security_group_ids = ["${aws_security_group.default.id}"]

  subnet_id = "${aws_subnet.default.id}"


  provisioner "file" {
    source = "${var.provisioning-folder}/"
    destination = "/tmp/vagrant"
  }

  provisioner "remote-exec" {
    scripts = [
      "${var.provisioning-folder}/cloud-vagrant-glue.sh",
      "${var.provisioning-folder}/base-provisioning.sh",
      "${var.provisioning-folder}/apache2-provisioning.sh",
      "${var.provisioning-folder}/nagios-client.sh"
    ]
  }


  connection {
    type     = "ssh"
    user     = "admin"
    private_key = "${file("${var.provisioning-folder}/provisioning-key")}"
  }
}
```

Otros scripts importantes.

Scan-local-network-for-nodes.sh (servidor maestro)

Teniendo que el número de nodos disponibles es variable y que hay que realizar configuración específica en cada uno de ellos para Haproxy y Nagios, tenemos un cron que se ejecuta cada 5 minutos que busca hosts en nuestra red local que tengan ambos el servicio SSH disponible y el servicio de agente de Nagios iniciado. Si los encuentra intenta conectarse con el usuario *puppet* y nuestra clave privada e inicia la configuración necesaria para añadirlos al servidor maestro.

```
#!/bin/bash
ownIP=`ip -o -4 addr | grep -oE "\b([0-9]{1,3}\.){3}[0-9]{1,3}" | grep -v -E "*255$" | grep -v
10.0.2.15 | grep -v 255.255.255.0 | grep -v 127.0.0.1 | grep -v 255.0.0.0`
netmask="/24"
localNetwork="$ownIP$netmask"
sudo nmap -sF -p 22,5666 $localNetwork | grep -B 5 "5666/tcp open|filtered nrpe" | grep -B 4
-A 1 "22/tcp   open|filtered ssh" | awk '/^Nmap scan report for/{print $5}' >
/tmp/hosts-with-ssh-open.list
sudo sed -i "/$ownIP/d" /tmp/hosts-with-ssh-open.list
while IFS= read -r potentialNodeIP
do
  grep  "$potentialNodeIP" /tmp/active-nodes.list
  if ! grep -q "$potentialNodeIP" /tmp/active-nodes.list;
       then
              echo "Trying node: $potentialNodeIP..."
              ssh -n -o BatchMode=yes -o ConnectTimeout=5 -o StrictHostKeyChecking=no -o
UserKnownHostsFile=/dev/null -i /home/puppet/.ssh/id_rsa puppet@$potentialNodeIP 'exit' &&
/bin/add-node.sh $potentialNodeIP $potentialNodeIP && echo $potentialNodeIP >>
/tmp/active-nodes.list
  fi

done < "/tmp/hosts-with-ssh-open.list"

while IFS= read -r nodeIP
do
       echo "Testing node: $nodeIP..."
       if ! ping -c 1 $nodeIP -W 1;
              then # Node is down
                     /bin/remove-node.sh $nodeIP
                     grep -v $nodeIP /tmp/active-nodes.list > /tmp/active-nodes.list.tmp
                     mv /tmp/active-nodes.list.tmp /tmp/active-nodes.list
  fi
done < "/tmp/active-nodes.list"
```

Terraform-node-generator.sh

```
#!/bin/bash

nodeBaseName="node"

totalNumberOfNodes=$1

nodeCount=1
tmpFiles="/tmp/tfnodes"
cd $tmpFiles

rm -rf $tmpFiles/_node*

while [ $nodeCount -le $totalNumberOfNodes ]
do
        currentNodeName=$nodeBaseName$nodeCount
        currentNodeFilename=$currentNodeName".tf"
        cat header.tf.template | sed "s/#INSTANCE-NAME#/$currentNodeName/g" >>
$tmpFiles/_$currentNodeFilename
        cat node.tf.template | sed "s/#INSTANCE-NAME#/$currentNodeName/g" >>
$tmpFiles/_$currentNodeFilename
        cat footer.tf.template | sed "s/#INSTANCE-NAME#/$currentNodeName/g" >>
$tmpFiles/_$currentNodeFilename
        nodeCount=$((nodeCount+1))
done

cd $tmpFiles
terraform apply
```

Nagios-add-host.sh (servidor maestro)

Dado que tenemos que añadir el servidor de Nagios a la whitelist de los agentes, *y que no necesariamente disponemos de esa IP en el momento del despliegue*, lo hacemos como parte del escaneado de nodos.

```
#!/bin/bash
hostIP=$1
hostname=$2
ownIP=`ip -o -4 addr | grep -oE "\b([0-9]{1,3}\.){3}[0-9]{1,3}" | grep -v -E "*255$" | grep -v
10.0.2.15 | grep -v 255.255.255.0 | grep -v 127.0.0.1 | grep -v 255.0.0.0`
sudo cp  /etc/nagios3/conf.d/nodes.template /etc/nagios3/conf.d/$hostname.cfg
sudo sed -i "s/#NODE-HOSTNAME#/$hostname/" /etc/nagios3/conf.d/$hostname.cfg
sudo sed -i "s/#NODE-IP#/$hostIP/" /etc/nagios3/conf.d/$hostname.cfg
echo "#!/bin/bash" > /tmp/addHost.sh
echo "sed -i \"s/.*allowed_hosts.*/allowed_hosts=127.0.0.1,$ownIP/\" /etc/nagios/nrpe.cfg" >>
/tmp/addHost.sh
echo "service nagios-nrpe-server restart" >> /tmp/addHost.sh
chmod +x /tmp/addHost.sh
scp -o BatchMode=yes -o ConnectTimeout=10 -o StrictHostKeyChecking=no -o
UserKnownHostsFile=/dev/null -i /home/puppet/.ssh/id_rsa /tmp/addHost.sh puppet@$hostIP:/tmp/
```

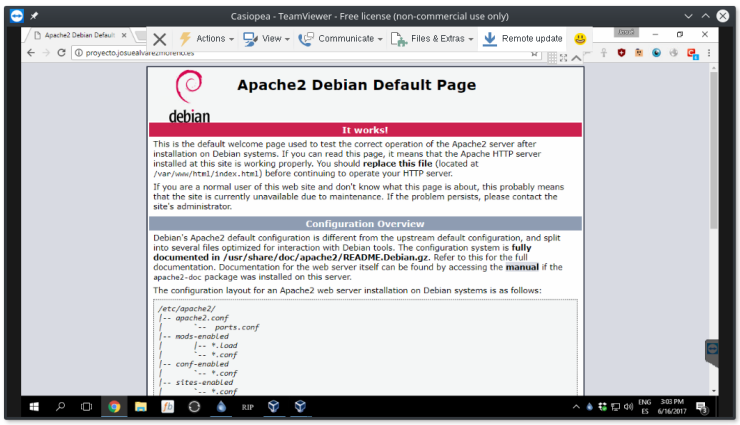
```
ssh -n -o BatchMode=yes -o ConnectTimeout=10 -o StrictHostKeyChecking=no -o
UserKnownHostsFile=/dev/null -i /home/puppet/.ssh/id_rsa puppet@$hostIP "chmod +x
/tmp/addHost.sh"
ssh -n -o BatchMode=yes -o ConnectTimeout=10 -o StrictHostKeyChecking=no -o
UserKnownHostsFile=/dev/null -i /home/puppet/.ssh/id_rsa puppet@$hostIP "sudo bash -c
/tmp/addHost.sh"
ssh -n -o BatchMode=yes -o ConnectTimeout=10 -o StrictHostKeyChecking=no -o
UserKnownHostsFile=/dev/null -i /home/puppet/.ssh/id_rsa puppet@$hostIP "sudo
/etc/init.d/nagios-nrpe-server restart"


sudo systemctl reload nagios3.service
```

Prueba de funcionamiento

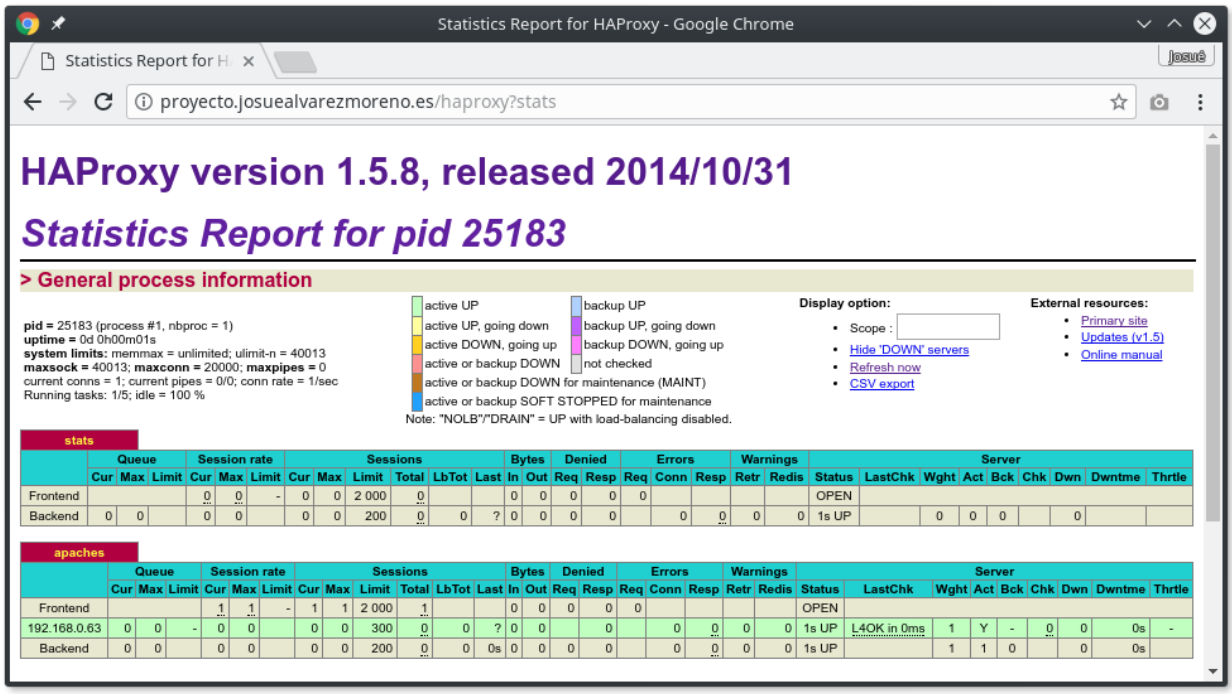
Levantamos el servidor maestro de Amazon, y nuestro *centro de datos de Vagrant*.

Comprobamos que el DNS apunta a nuestro centro de datos, y que es accesible desde otra red diferente a la nuestra:

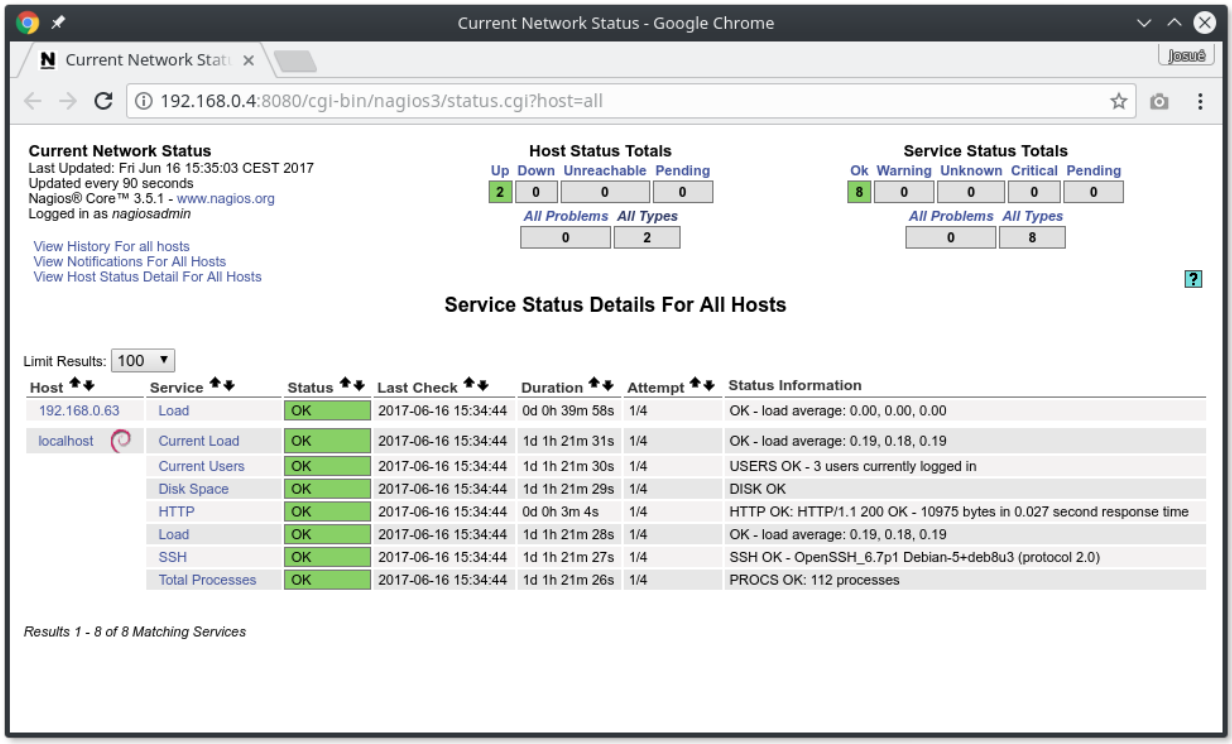
```
dig @8.8.8.8 +short proyecto.josuealvarezmoreno.es
217.217.241.49
```



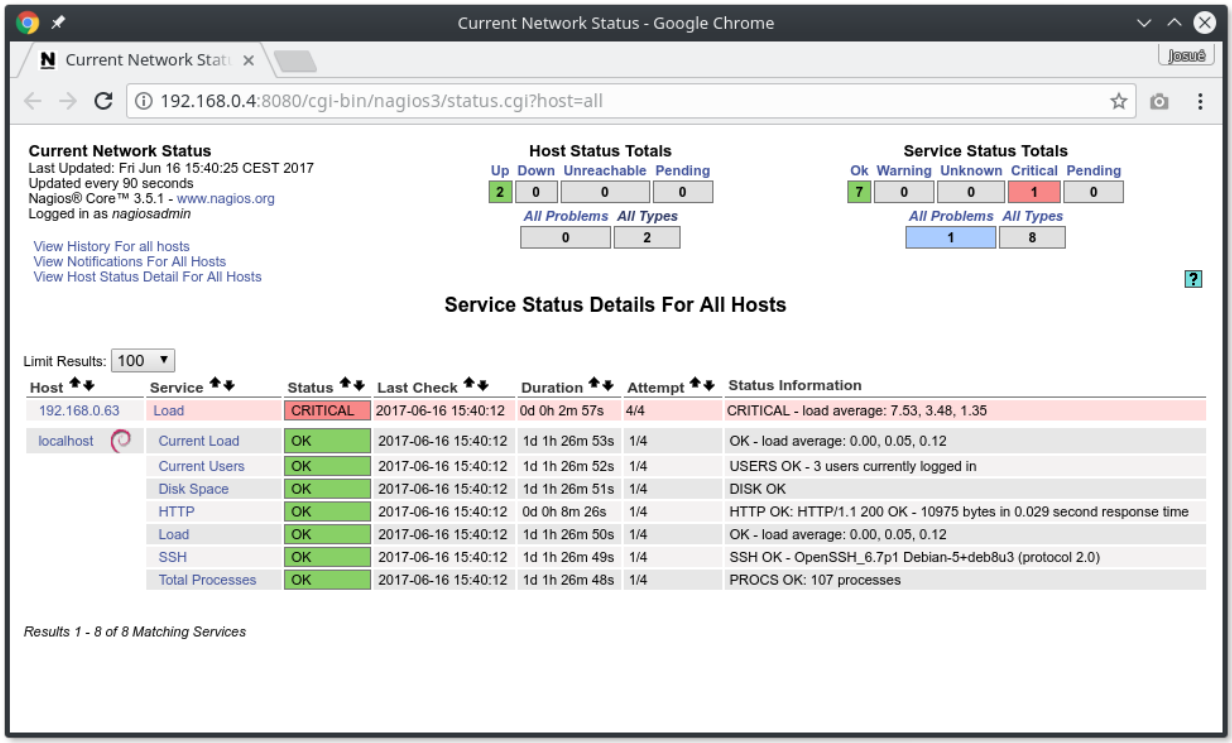
Nuestro Haproxy:



Y Nagios:



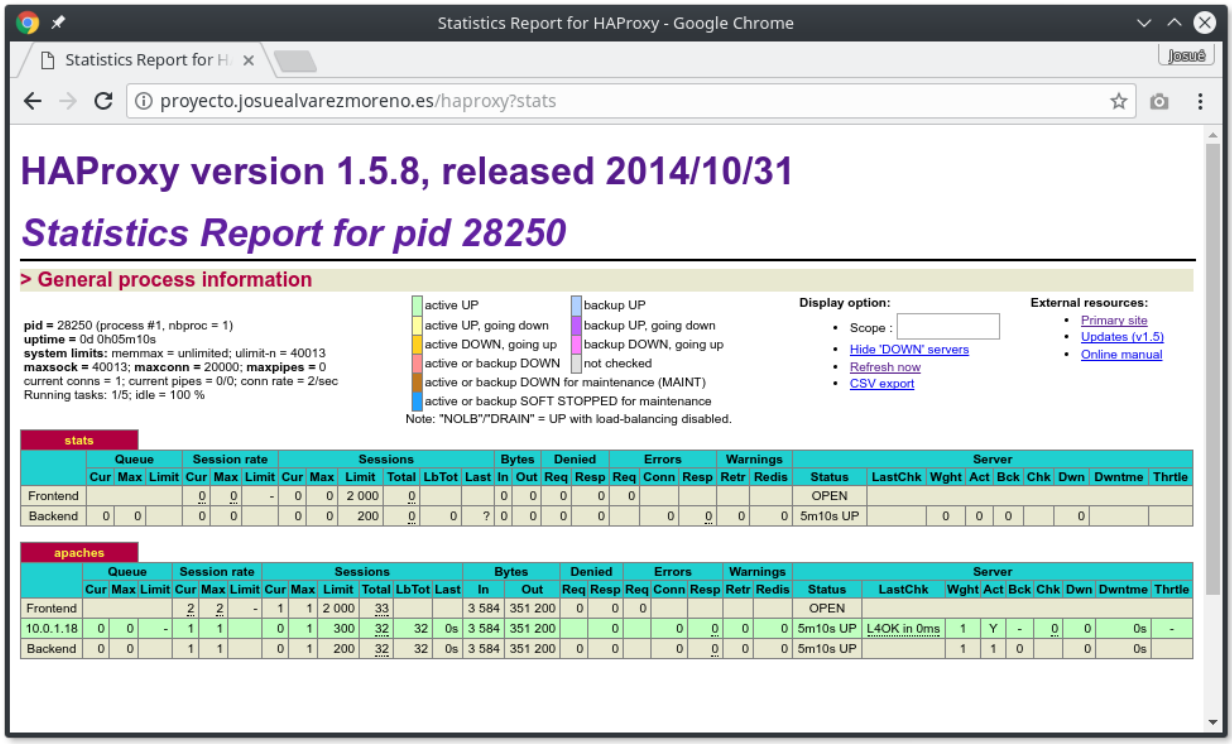
Vamos a generar carga de CPU con stress -c 8, y veamos qué ocurre.

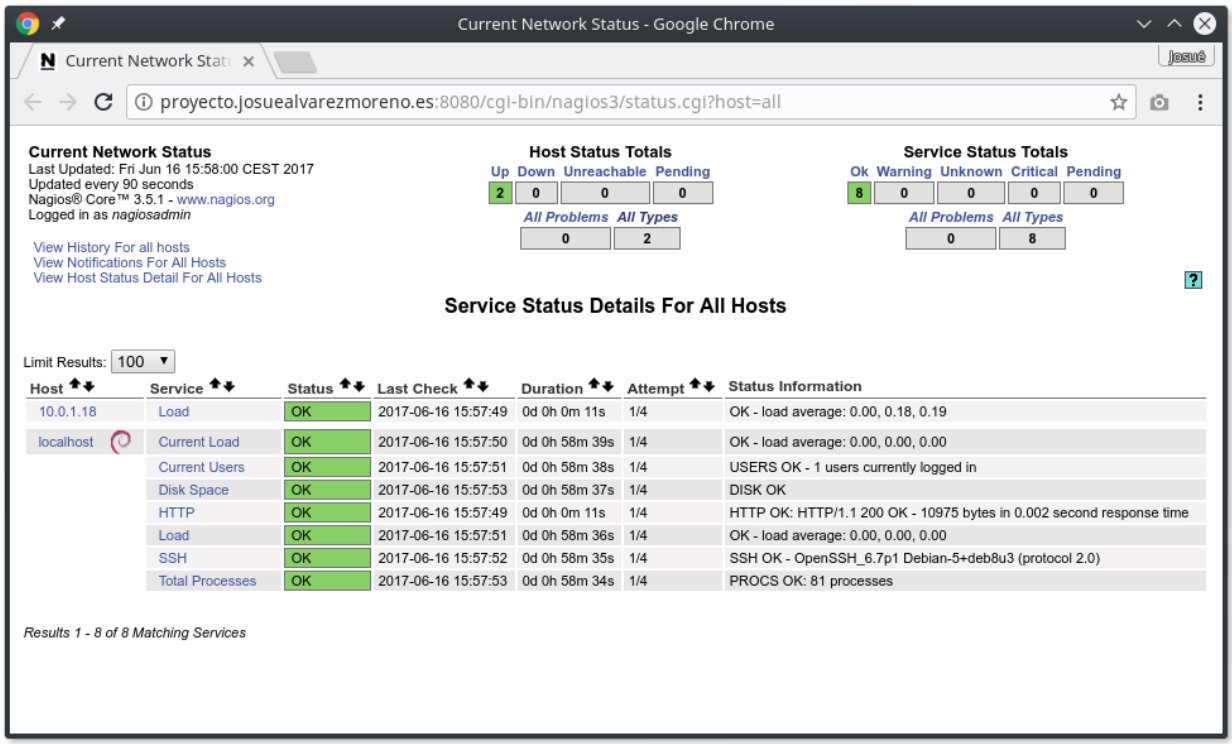


Si comprobamos los logs del servidor maestro en AWS, veremos que está generando el nodo1:

```
aws_key_pair.ssh-keys: Refreshing state... (ID: terraform)
aws_vpc.default: Refreshing state... (ID: vpc-70ba1409)
aws_internet_gateway.default: Refreshing state... (ID: igw-b18911d7)
aws_subnet.default: Refreshing state... (ID: subnet-3cb01a10)
aws_security_group.elb: Refreshing state... (ID: sg-ed14ec9c)
aws_security_group.default: Refreshing state... (ID: sg-ec17ef9d)
aws_route.internet_access: Refreshing state... (ID:
r-rtb-f8bfd4801080289494)
aws_instance.main: Refreshing state... (ID: i-03e6adda519fb0b0c)
aws_instance.node1: Creating...
  ami:                                "" => "ami-b14ba7a7"
  associate_public_ip_address:        "" => "<computed>"
  availability_zone:                  "" => "us-east-1a"
  ebs_block_device.#:                 "" => "<computed>"
  ephemeral_block_device.#:           "" => "<computed>"
  instance_state:                     "" => "<computed>"
  instance_type:                      "" => "t2.micro"
  ipv6_address_count:                 "" => "<computed>"
  ipv6_addresses.#:                   "" => "<computed>"
  key_name:                           "" => "terraform"
  network_interface.#:                "" => "<computed>"
  network_interface_id:               "" => "<computed>"
  placement_group:                    "" => "<computed>"
  primary_network_interface_id:       "" => "<computed>"
  private_dns:                        "" => "<computed>"
  private_ip:                         "" => "<computed>"
  public_dns:                         "" => "<computed>"
  public_ip:                          "" => "<computed>"
  root_block_device.#:                "" => "<computed>"
  security_groups.#:                  "" => "<computed>"
  source_dest_check:                  "" => "true"
  subnet_id:                          "" => "subnet-3cb01a10"
  tenancy:                            "" => "<computed>"
  volume_tags.%:                      "" => "<computed>"
  vpc_security_group_ids.#:           "" => "1"
  vpc_security_group_ids.3058249602: "" => "sg-ec17ef9d"
aws_instance.node1: Still creating... (10s elapsed)
aws_instance.node1: Still creating... (20s elapsed)
```

4 minutos después:

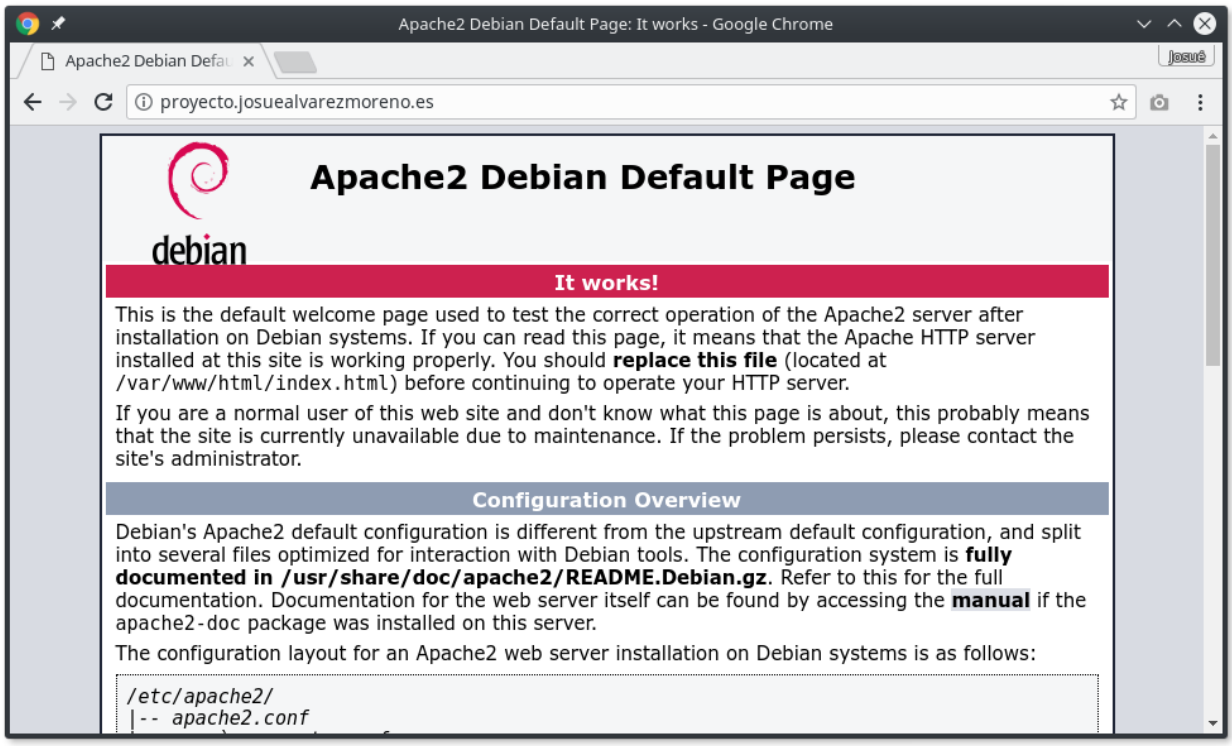

Statistics Report for HAProxy - Google Chrome
Statistics Report for H
proyecto.josuealvarezmoreno.es/haproxy?stats
HAProxy version 1.5.8, released 2014/10/31
Statistics Report for pid 28250
> General process information
pid = 28250 (process #1, nbproc = 1)
uptime = 0d 0h05m10s
system limits: memmax = unlimited; ulimit-n = 40013
maxsock = 40013; maxconn = 20000; maxpipes = 0
current conns = 1; current pipes = 0/0; conn rate = 2/sec
Running tasks: 1/5; idle = 100 %
active UP
active UP, going down
active DOWN, going up
active or backup DOWN
active or backup DOWN for maintenance (MAINT)
active or backup SOFT STOPPED for maintenance
backup UP
backup UP, going down
backup DOWN, going up
not checked
Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.
Display option:
Scope :
Hide 'DOWN' servers
Refresh now
CSV export
External resources:
Primary site
Updates (v1.5)
Online manual
stats
Queue
Session rate
Sessions
Bytes
Denied
Errors
Warnings
Server
Cur Max Limit Cur Max Limit Cur Max Limit Total LbTot Last In Out Req Resp Req Conn Resp Retr Redis Status LastChk Wght Act Bck Chk Dwn Dwntme Thrtle
Frontend 0 0 - 0 0 2 000 0 OPEN
Backend 0 0 0 0 0 0 200 0 0 ? 0 0 0 0 0 0 0 0 5m10s UP 0 0 0 0 0
apaches
Queue
Session rate
Sessions
Bytes
Denied
Errors
Warnings
Server
Cur Max Limit Cur Max Limit Cur Max Limit Total LbTot Last In Out Req Resp Req Conn Resp Retr Redis Status LastChk Wght Act Bck Chk Dwn Dwntme Thrtle
Frontend 2 2 - 1 1 2 000 33 3 584 351 200 0 0 0 OPEN
10.0.1.18 0 0 - 1 1 0 1 300 32 32 0s 3 584 351 200 0 0 0 0 0 5m10s UP L4OK in 0ms 1 Y - 0 0 0s -
Backend 0 0 1 1 0 1 200 32 32 0s 3 584 351 200 0 0 0 0 0 0 0 5m10s UP 1 1 0 0 0s


Current Network Status - Google Chrome
Current Network Stat
proyecto.josuealvarezmoreno.es:8080/cgi-bin/nagios3/status.cgi?host=all
Current Network Status
Last Updated: Fri Jun 16 15:58:00 CEST 2017
Updated every 90 seconds
Nagios® Core™ 3.5.1 - www.nagios.org
Logged in as nagiosadmin
View History For all hosts
View Notifications For All Hosts
View Host Status Detail For All Hosts
Host Status Totals
Up Down Unreachable Pending
2 0 0 0
All Problems All Types
0 2
Service Status Totals
Ok Warning Unknown Critical Pending
8 0 0 0 0
All Problems All Types
0 8
Service Status Details For All Hosts
Limit Results: 100
Host Service Status Last Check Duration Attempt Status Information
10.0.1.18 Load OK 2017-06-16 15:57:49 0d 0h 0m 11s 1/4 OK - load average: 0.00, 0.18, 0.19
localhost Current Load OK 2017-06-16 15:57:50 0d 0h 58m 39s 1/4 OK - load average: 0.00, 0.00, 0.00
Current Users OK 2017-06-16 15:57:51 0d 0h 58m 38s 1/4 USERS OK - 1 users currently logged in
Disk Space OK 2017-06-16 15:57:53 0d 0h 58m 37s 1/4 DISK OK
HTTP OK 2017-06-16 15:57:49 0d 0h 0m 11s 1/4 HTTP OK: HTTP/1.1 200 OK - 10975 bytes in 0.002 second response time
Load OK 2017-06-16 15:57:51 0d 0h 58m 36s 1/4 OK - load average: 0.00, 0.00, 0.00
SSH OK 2017-06-16 15:57:52 0d 0h 58m 35s 1/4 SSH OK - OpenSSH_6.7p1 Debian-5+deb8u3 (protocol 2.0)
Total Processes OK 2017-06-16 15:57:53 0d 0h 58m 34s 1/4 PROCS OK: 81 processes
Results 1 - 8 of 8 Matching Services

Vemos que el DNS ya se ha actualizado.

```
$ dig @8.8.8.8 +short proyecto.josuealvarezmoreno.es
54.210.100.60
```

Y listo. En vez de un único nodo, podríamos configurarlo para que lance 8 (o 16, o 24) pero para las pruebas es suficiente.



Preguntas que te estás haciendo ahora mismo, y posibles mejoras.

Nagios está monitorizando el localhost, ¿por qué?

Nagios no arranca si no tiene ningún host que monitorizar. Como la idea del servidor maestro de nagios es que esté siempre disponible, para poder encontrar nodos conforme vayan arrancando, he dejado la configuración del localhost.

¿Por qué no conectas los nodos a tu centro de datos? Te ahorras una instancia.

Totalmente de acuerdo. En lugar de tener el servidor maestro en la nube, debería alojarlo en nuestro centro de datos y hacer que los nodos se conecten a él mediante VPN o incluso túneles SSH. No hay *necesidad* de tener el maestro alojado externamente.
Una segunda iteración de este diseño lo hubiese incluído, ya que además nos permite *bridgear* el servidor web de la nube con una base de datos en nuestro CPD.

Eso, ¿y la base de datos?

Depende de qué software base de datos estemos usando, lo tenemos difícil o *ligeramente menos difícil*. También tenemos que tener en cuenta que tipo de HA deseamos: Escrituras garantizadas, rendimiento por encima de todo, consistencia eventual…

No es lo mismo un blog o una red social, en el que podemos permitirnos refrescar cada ciertos minutos… que un enorme portal de ventas en el que cada transacción (tanto monetaria como en términos de base de datos) es vital e irremplazable.

Y hemos de evaluar también:

- La naturaleza de las consultas: Un blog con comentarios, en el que la query es hacer unos select con unas cláusulas where? Un modelo tipo *feed* (como en Twitter/Facebook) en el que la query es hacer una select de todos los posts de cada persona a la que sigo, posiblemente hacer un join o dos, eliminar la gente a la que sigo pero tengo silenciada, eliminar los posts que sean reposts de gente a la que tengo bloqueada…?
- Y, **legalmente muy importante**: Información de carácter Personal y las restricciones legales aplicables. ¿Podemos exportar la base de datos fuera de mi centro de datos, y en qué condiciones? ¿Son la nube que voy a usar y los datos que voy a exportar compatibles con la legislación de España? ¿Y la europea?

- Suponiendo que puedas exportar la base de datos *hacia fuera* sin problemas y hacer varios nodos (o incluso *sharding*), ¿cuál es la estrategia para traerla de vuelta manteniendo la Alta Disponibilidad?

El ámbito de esta sección podría ocupar otras quince páginas, y lamentablemente no me ha dado tiempo de acercarme siquiera.

¿Cómo de-escalar de la nube a nuestro centro de datos?

Es un paso que realizaría manualmente. Expandirse automáticamente significa que puedes pagar cien dólares extra. De-escalar automáticamente significa que en caso de error estás sirviendo 503s a todo el mundo, y es justo lo contrario de lo que queremos conseguir.

Dicho esto, conceptualmente es sencillo:

- Si la web es estática, no hay que almacenar el estado y podemos saltar a modificar el dns.
- Si tenemos base de datos, y la hemos exportado al exterior: Estudiar la mejor estrategia para traernos las diferencias a nuestro centro de datos.
- Es posible que interrumpamos las sesiones de los usuarios si dependen de variables en el servidor, hay que tenerlo en cuenta.
- ddclient en tu centro de datos para modificar el DNS a la IP que corresponde.
- Esperar un tiempo prudencial a que se propague el cambio (aunque tengas un TTL de 1 minuto).
- *terraform destroy*.

¡Mi página web no es “*It Works*!”, se trata de una página web estática pero de verdad con enlaces, secciones e información.

Podrías realizar un *rsync* al master y que éste provisione a los nodos antes de añadirlos a Haproxy. O si tienes tu página web almacenada en un sistema de control de versiones, puedes añadir el equivalente a *git clone*.