

2017

Geolocalizador de IPs

Práctica Cloud Computing

Máster Ingeniería Informática
Universidad de Huelva



Índice

Introducción	3
Introducción a nuestro sistema.....	3
Implementación del sistema	3
Pruebas del sistema.....	5
Configuración inicial	5
Primera prueba (2 servidores inicialmente)	6
Segunda prueba (4 servidores inicialmente)	7
Tercera prueba (5 servidores inicialmente).....	9
Resultados obtenidos	11

Introducción

Se ha de desarrollar un sistema de Geolocalización de IPs capaz de atender a cientos de miles de peticiones por minuto.

Para ello hemos hecho uso de **Amazon AWS** para crear un servicio auto escalable y de alta disponibilidad, creando un sistema de **balanceador de carga con auto-escalado** explicado paso a paso en la teoría.

Introducción a nuestro sistema

Nuestro sistema de geolocalización de IPs consta de un servicio API REST haciendo uso de **Node.JS con la BBDD precargada en la memoria RAM** en un Ubuntu Server 16.04.

Se ha hecho uso de Node.JS ya que resuelve las peticiones de manera asíncrona en el servidor y, por tanto, es el adecuado cuando necesitas resolver y hacer muchas cosas al mismo tiempo.

Se ha optado por precargar la BBDD en la memoria RAM del servidor para aprovechar la gran velocidad de respuesta de esta ya que la BBDD es estática.

Implementación del sistema

Una vez instalado el Ubuntu Sever 16.04, procedemos a instalar Node.JS para ello puede seguir el siguiente tutorial:

¿Cómo instalar Node.js en Ubuntu 16.04?

<https://www.digitalocean.com/community/tutorials/como-instalar-node-js-en-ubuntu-16-04-es>

Como nuestro sistema es de auto-escalado, necesitamos que nuestro servidor Node.JS se ejecute automáticamente al inicio del servidor. Para ello hacemos uso del paquete de Node.JS llamado PM2, los pasos para lograrlo se pueden encontrar en:

** Es necesario crear primero nuestra propia aplicación Node.JS*

How To Use PM2 to Setup a Node.js Production Environment On An Ubuntu VPS

<https://www.digitalocean.com/community/tutorials/how-to-use-pm2-to-setup-a-node-js-production-environment-on-an-ubuntu-vps>

Implementamos nuestra API REST con nuestra propia aplicación Node.JS con el código mostrado en la siguiente página, destacar el uso del Framework Node Express para facilitar la programación y la librería ip2loc para precargar la BBDD en memoria y realizar las consultas a esta.

```

var express = require("express"),
    app = express(),
    bodyParser = require("body-parser"),
    methodOverride = require("method-override"),
    ip2loc = require("ip2location-nodejs"),
    js2xmlparser = require("js2xmlparser"),
    json2csv = require('json2csv');

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
app.use(methodOverride());

var router = express.Router();

router.get('/', function(req, res) {
    res.send("Hello World!");
});

router.get('/json/:ip', function(req, res) {
    var ip = req.params.ip;

    if (ip == "random") {
        ip = (Math.floor(Math.random() * (255 - 1)) + 1)+"."+(Math.floor(Math.random() * (255 - 1)) + 1)+"."
        +(Math.floor(Math.random() * (255 - 1)) + 1)+"."+(Math.floor(Math.random() * (255 - 1)) + 1);
    }

    temp = ip2loc.IP2Location_get_all(ip);

    res.send(temp);
});

router.get('/xml/:ip', function(req, res) {
    var ip = req.params.ip;

    if (ip == "random") {
        ip = (Math.floor(Math.random() * (255 - 1)) + 1)+"."+(Math.floor(Math.random() * (255 - 1)) + 1)+"."
        +(Math.floor(Math.random() * (255 - 1)) + 1)+"."+(Math.floor(Math.random() * (255 - 1)) + 1);
    }

    temp = ip2loc.IP2Location_get_all(ip);

    res.send(js2xmlparser.parse("Response", temp));
});

router.get('/csv/:ip', function(req, res) {
    var ip = req.params.ip;

    if (ip == "random") {
        ip = (Math.floor(Math.random() * (255 - 1)) + 1)+"."+(Math.floor(Math.random() * (255 - 1)) + 1)+"."
        +(Math.floor(Math.random() * (255 - 1)) + 1)+"."+(Math.floor(Math.random() * (255 - 1)) + 1);
    }

    temp = ip2loc.IP2Location_get_all(ip);

    res.send(js2xmlparser.parse("Response", temp));
});

app.use(router);

app.listen(80, function() {
    console.log("Node server running on http://localhost:80");
});

ip2loc.IP2Location_init("IP2LOCATION-LITE-DB11.BIN");

```

Pruebas del sistema

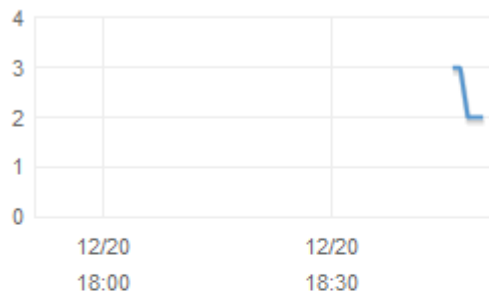
Configuración inicial

Se va a iniciar la configuración del balanceador de carga con un mínimo de 2 servidores hasta un máximo de 5 servidores, que irán aumentando o disminuyendo según se indicó en las instrucciones de la práctica.

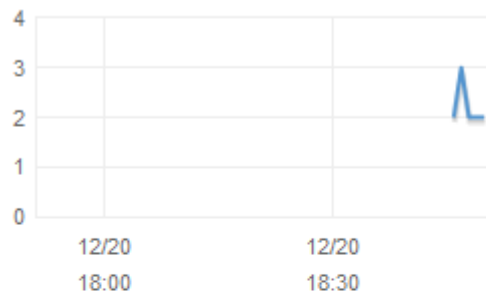
Name ▲	Launch Configuration ▼	Instances ▼	Desired ▼	Min ▼	Max ▼
sm-ap-aeg	sm-ap-lc	2	2	2	5

Podemos observar en las gráficas como, antes de realizar las pruebas, disponemos de 2 servidores resolviendo peticiones.

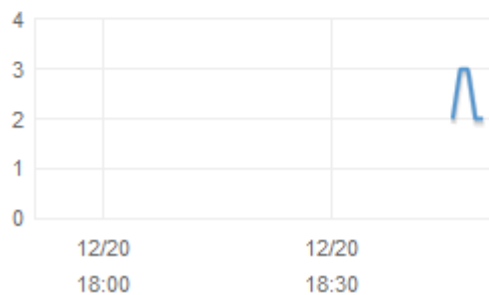
Desired Capacity (Count)



In Service Instances (Count)



Total Instances (Count)



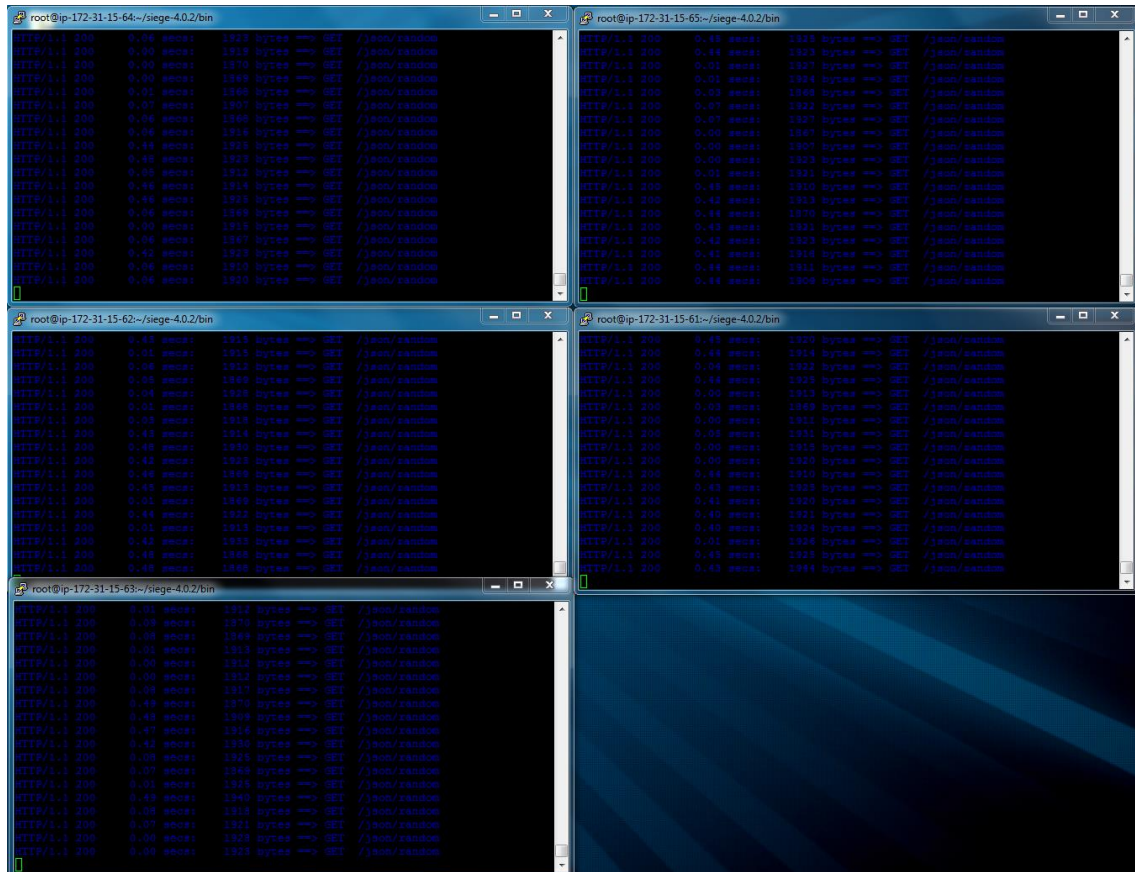
Para realizar las pruebas se usarán 5 clientes con la aplicación siege lanzando 400 peticiones concurrentes durante 5 minutos en cada prueba, usando el comando:

```
./siege -c 400 -t 5m sm-ap-lc-52955897.eu-central-1.elb.amazonaws.com/json/random
```

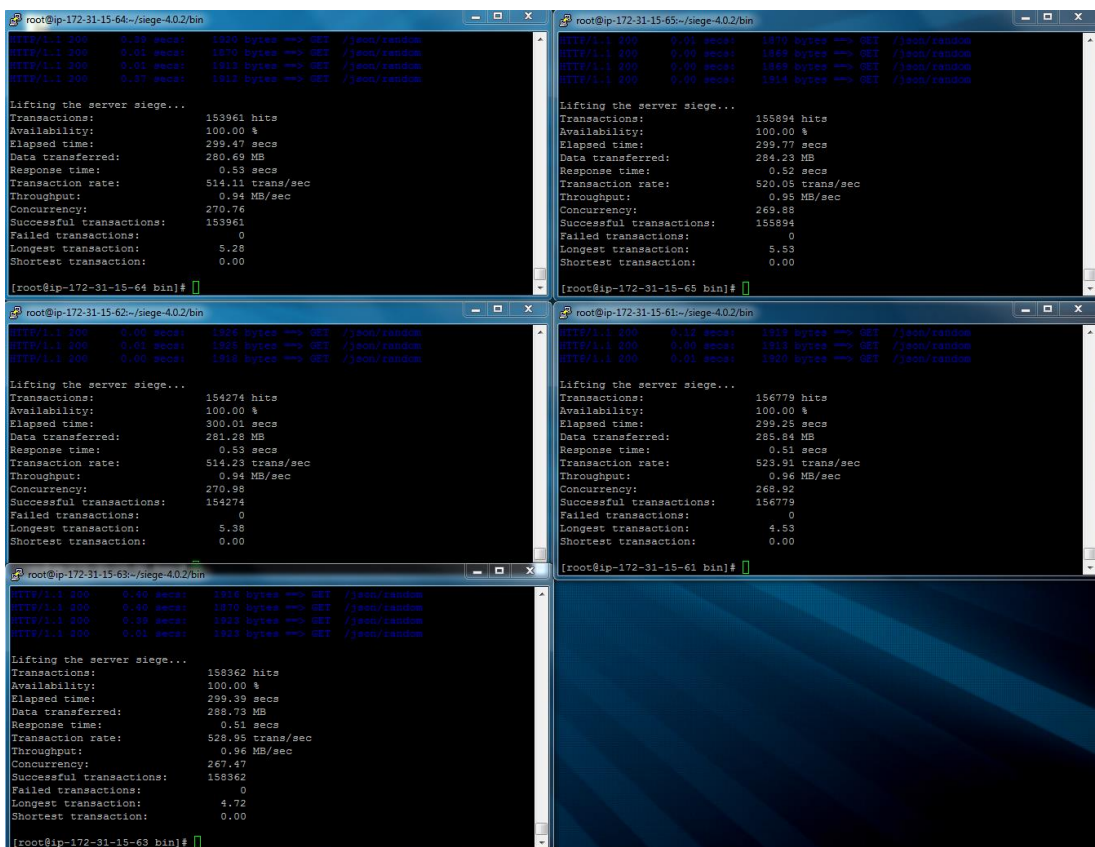
Primera prueba (2 servidores inicialmente)

Comenzamos la primera prueba con 2 servidores inicialmente contestando a las peticiones, como hemos indicado en la configuración inicial.

Lanzamos el comando de siege en los 5 clientes



Una vez transcurridos los 5 minutos de 400 peticiones concurrentes, obtenemos el siguiente resultado:

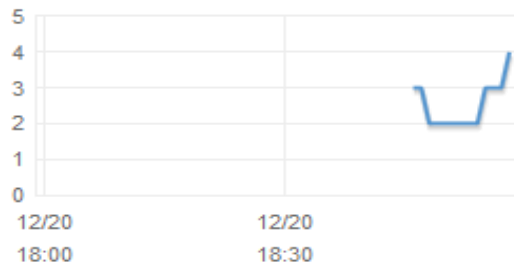


Si nos fijamos en el balanceador de carga, obtenemos que ha tenido que subir el número de servidores que resuelven las peticiones a 4, aunque como veremos en las gráficas las pruebas terminaron con 3 servidores resolviendo peticiones.

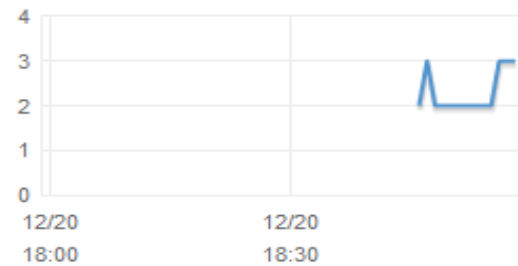
<input type="checkbox"/>	Name	Launch Configuration	Instances	Desired	Min	Max
<input checked="" type="checkbox"/>	sm-ap-aeg	sm-ap-lc	4	4	2	5

Con las siguientes gráficas:

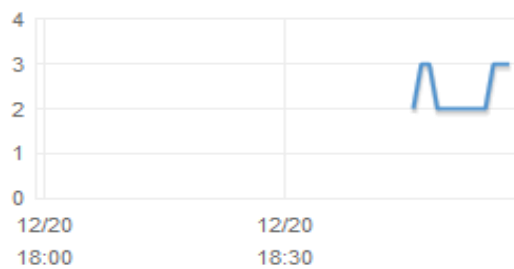
Desired Capacity (Count)



In Service Instances (Count)



Total Instances (Count)



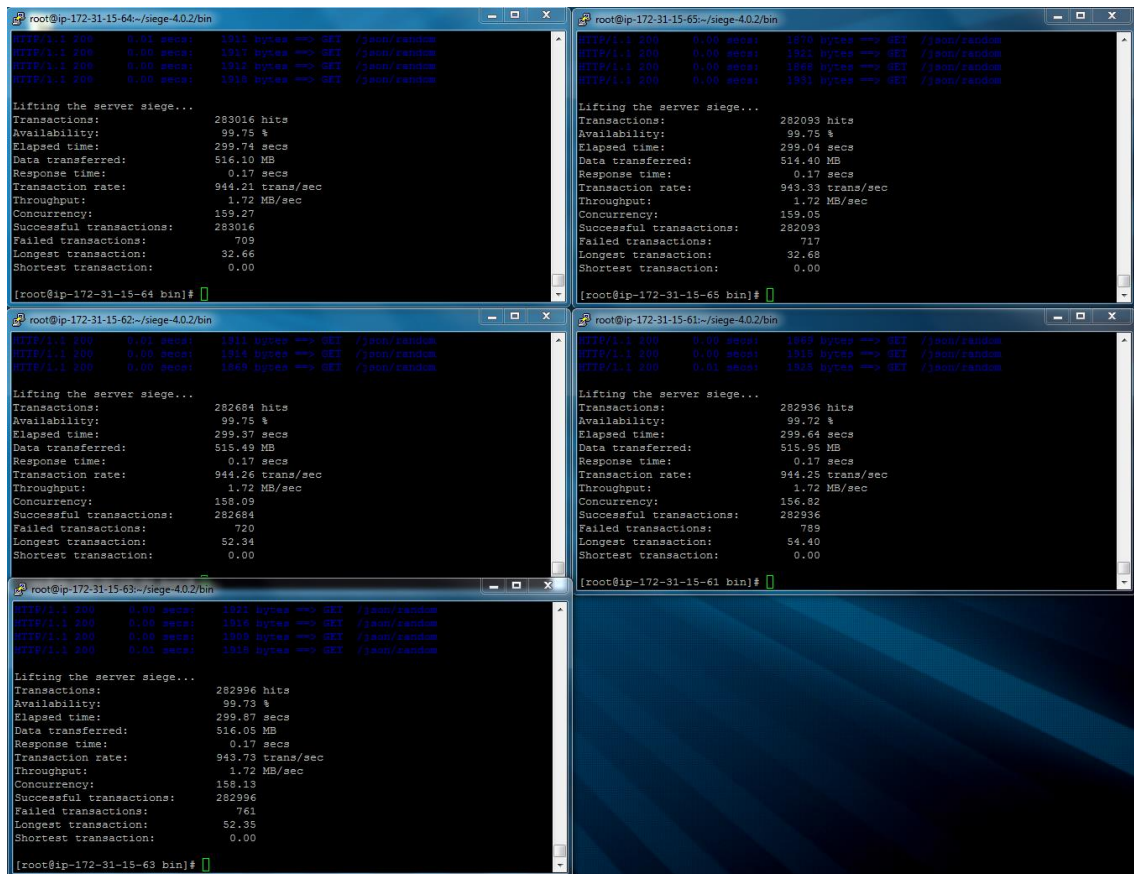
Observamos cómo, transcurrido el tiempo, el balanceador de carga a tenido que ir aumentando el número de instancias que resuelven peticiones desde 2 hasta 4.

Cuando terminan las pruebas aún no se ha iniciado la cuarta instancia, por lo que solo se han respondido peticiones desde 3 servidores al finalizar la prueba.

Segunda prueba (4 servidores inicialmente)

Lanzamos la segunda prueba con 4 servidores inicialmente contestando a las peticiones esta vez.

Lanzamos el comando de siege en los 5 clientes, una vez transcurridos los 5 minutos de 400 peticiones concurrentes, obtenemos el siguiente resultado:



Podemos destacar como casi se ha duplicado el número de transacciones por segundo, estos resultados se detallarán y contrastarán en el apartado de resultados obtenidos.

Si nos fijamos en el balanceador de carga, obtenemos que ha tenido que subir el número de servidores que resuelven las peticiones a 5.

<input checked="" type="checkbox"/>	sm-ap-aeg	sm-ap-lc	5	5	2	5
-------------------------------------	-----------	----------	---	---	---	---

Con las siguientes gráficas:

Desired Capacity (Count)



In Service Instances (Count)



Total Instances (Count)



Observamos cómo, transcurrido el tiempo, el balanceador de cargar a tenido que ir aumentando el número de instancias que resuelven peticiones a 5.

Tercera prueba (5 servidores inicialmente)

Lanzamos la tercera prueba con los 5 servidores contestando a las peticiones esta vez.

Lanzamos el comando de siege en los 5 clientes, una vez transcurridos los 5 minutos de 400 peticiones concurrentes, obtenemos el siguiente resultado:

```
root@ip-172-31-15-64:~/siege-4.0.2/bin
HITP[1] 1 200 0.00 secs: 1024 bytes ==> GET /jawn/random
HITP[1] 1 200 0.00 secs: 1024 bytes ==> GET /jawn/random
HITP[1] 1 200 0.00 secs: 1024 bytes ==> GET /jawn/random
HITP[1] 1 200 0.00 secs: 1024 bytes ==> GET /jawn/random

Lifting the server siege...
Transactions: 307507 hits
Availability: 99.96 %
Elapsed time: 299.46 secs
Data transferred: 560.64 MB
Response time: 0.13 secs
Transaction rate: 1026.87 trans/sec
Throughput: 1.87 MB/sec
Concurrency: 137.69
Successful transactions: 307507
Failed transactions: 124
Longest transaction: 9.01
Shortest transaction: 0.00
[root@ip-172-31-15-64 bin]#

root@ip-172-31-15-65:~/siege-4.0.2/bin
HITP[1] 1 200 0.00 secs: 1016 bytes ==> GET /jawn/random
HITP[1] 1 200 0.00 secs: 1017 bytes ==> GET /jawn/random
HITP[1] 1 200 0.00 secs: 1016 bytes ==> GET /jawn/random
HITP[1] 1 200 0.00 secs: 1013 bytes ==> GET /jawn/random

Lifting the server siege...
Transactions: 307655 hits
Availability: 99.97 %
Elapsed time: 299.73 secs
Data transferred: 560.91 MB
Response time: 0.13 secs
Transaction rate: 1026.44 trans/sec
Throughput: 1.87 MB/sec
Concurrency: 138.02
Successful transactions: 307655
Failed transactions: 105
Longest transaction: 8.03
Shortest transaction: 0.00
[root@ip-172-31-15-65 bin]#

root@ip-172-31-15-62:~/siege-4.0.2/bin
HITP[1] 1 200 0.00 secs: 1016 bytes ==> GET /jawn/random
HITP[1] 1 200 0.00 secs: 1010 bytes ==> GET /jawn/random
HITP[1] 1 200 0.00 secs: 1004 bytes ==> GET /jawn/random

Lifting the server siege...
Transactions: 307586 hits
Availability: 99.96 %
Elapsed time: 299.93 secs
Data transferred: 561.25 MB
Response time: 0.13 secs
Transaction rate: 1026.43 trans/sec
Throughput: 1.87 MB/sec
Concurrency: 136.44
Successful transactions: 307586
Failed transactions: 165
Longest transaction: 8.53
Shortest transaction: 0.00
[root@ip-172-31-15-62 bin]#

root@ip-172-31-15-61:~/siege-4.0.2/bin
HITP[1] 1 200 0.00 secs: 1018 bytes ==> GET /jawn/random
HITP[1] 1 200 0.00 secs: 1010 bytes ==> GET /jawn/random
HITP[1] 1 200 0.01 secs: 1003 bytes ==> GET /jawn/random
HITP[1] 1 200 0.01 secs: 1000 bytes ==> GET /jawn/random

Lifting the server siege...
Transactions: 307553 hits
Availability: 99.96 %
Elapsed time: 299.23 secs
Data transferred: 560.31 MB
Response time: 0.14 secs
Transaction rate: 1027.05 trans/sec
Throughput: 1.87 MB/sec
Concurrency: 138.77
Successful transactions: 307523
Failed transactions: 110
Longest transaction: 9.07
Shortest transaction: 0.00
[root@ip-172-31-15-61 bin]#

root@ip-172-31-15-63:~/siege-4.0.2/bin
HITP[1] 1 200 0.00 secs: 1018 bytes ==> GET /jawn/random
HITP[1] 1 200 0.00 secs: 1010 bytes ==> GET /jawn/random
HITP[1] 1 200 0.01 secs: 1003 bytes ==> GET /jawn/random
HITP[1] 1 200 0.01 secs: 1000 bytes ==> GET /jawn/random

Lifting the server siege...
Transactions: 307350 hits
Availability: 99.97 %
Elapsed time: 299.28 secs
Data transferred: 560.33 MB
Response time: 0.13 secs
Transaction rate: 1026.96 trans/sec
Throughput: 1.87 MB/sec
Concurrency: 138.52
Successful transactions: 307350
Failed transactions: 99
Longest transaction: 11.35
Shortest transaction: 0.00
[root@ip-172-31-15-63 bin]#
```

Podemos destacar como solo ha subido ligeramente el número de transacciones por segundo, estos resultados se detallarán y contrastarán en el apartado de resultados obtenidos.

Si nos fijamos en el balanceador de carga, vemos como se mantiene el número de servidores que resuelven las peticiones en 5.

Name	Launch Configuration	Instances	Desired	Min	Max
sm-ap-aeg	sm-ap-lc	5	5	2	5

Con las siguientes gráficas:

Desired Capacity (Count)



In Service Instances (Count)



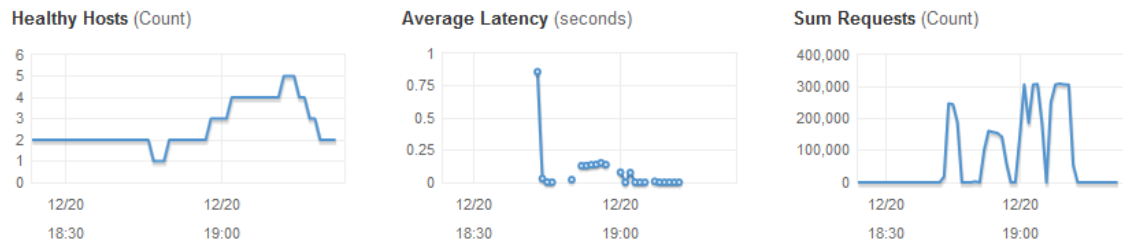
Total Instances (Count)



Observamos cómo, transcurrido el tiempo, el balanceador de carga a llegado un momento que bajo el número de servidores que resuelven peticiones a 4, pero al poco tiempo tuvo que volver a subirlos a 5.

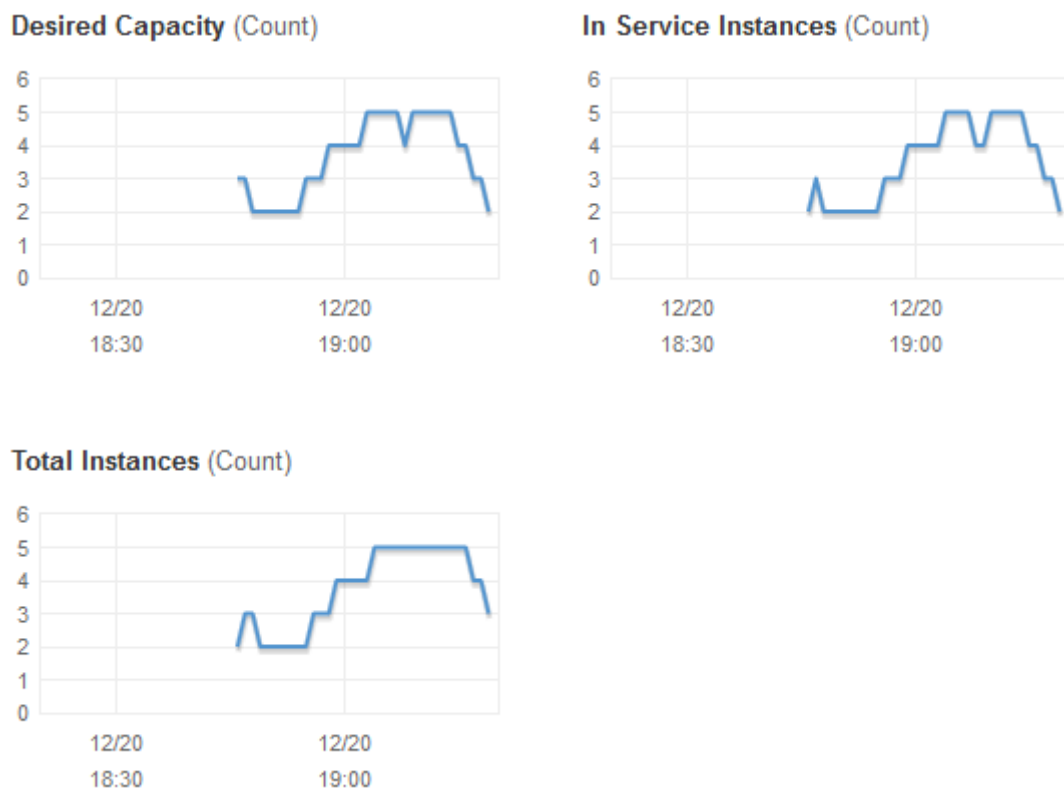
Resultados obtenidos

Comenzamos observando las gráficas obtenidas de la monitorización del balanceador de carga, destacando las siguientes 3 gráficas:

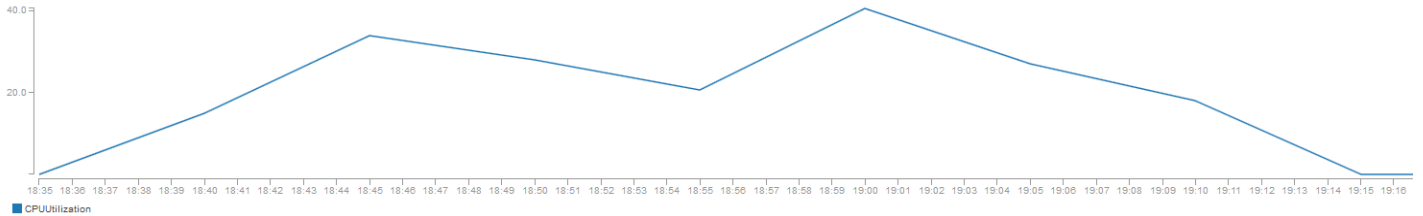


En ella obtenemos como ha incrementado y decrecido el número de instancias que resuelven peticiones, la latencia media está muy próxima a 0 y picos de más de 300.000 peticiones.

También podemos observar cómo, después de realizar las pruebas, el número de instancias decrece con el tiempo.



También podemos obtener cómo ha variado con el tiempo el porcentaje de utilización de CPU. Se puede observar como al incrementar el número de instancias que resuelven las peticiones, el porcentaje de utilización de CPU media baja ya que se reparte el trabajo en más instancias.



Creamos una tabla con los resultados obtenidos medios en cada prueba:

	Prueba 1 (2 Inst.)	Prueba 2 (4 Inst.)	Prueba 3 (5 Inst.)
Transacciones	157.000	282.800	307.500
Disponibilidad	100 %	99.75 %	99.96 %
Transacciones/Segundo	524	944	1027
MBs/Segundo	0.95	1.72	1.87
Concurrencia	270	158	138

Contrastando por orden según la tabla, obtenemos como incrementa, según el número de instancias, el número de transacciones que resuelven durante los 5 minutos de las pruebas.

La disponibilidad empeora al lanzar nuevas instancias, ya que el tiempo indicado al balanceador de carga para que indique una instancia como disponible, es menor que el tiempo que tarda ese servidor en iniciarse. Podemos corregir este problema incrementando ese tiempo.

Las Transacciones/Segundo y los MBs/Segundo vemos como incrementa bastante según el número de instancias resolviendo peticiones. Y como la concurrencia le ocurre lo contrario, al haber más ordenadores resolviendo peticiones.