Ingeniería de Servidores (2015-2016)
Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Memoria Práctica 4

Iván Sevillano García

12 de junio de 2016

Índice

1. Benchmarks populares

1.1. Phoronix tests suites

- **Instale la aplicación. ¿Qué comando permite listar los benchmarks disponibles?**
  Para instalar phoronix, en la página oficial nos recomiendan descargar el paquete .deb e instalarlo a partir del mismo[1].
  Según la documentación de phoronix [2], el comando para ver que benchmarks hay disponibles es el siguiente:
  phoronix-test-suite list-available-tests
  Algunos de los tests que están disponibles son los siguientes:

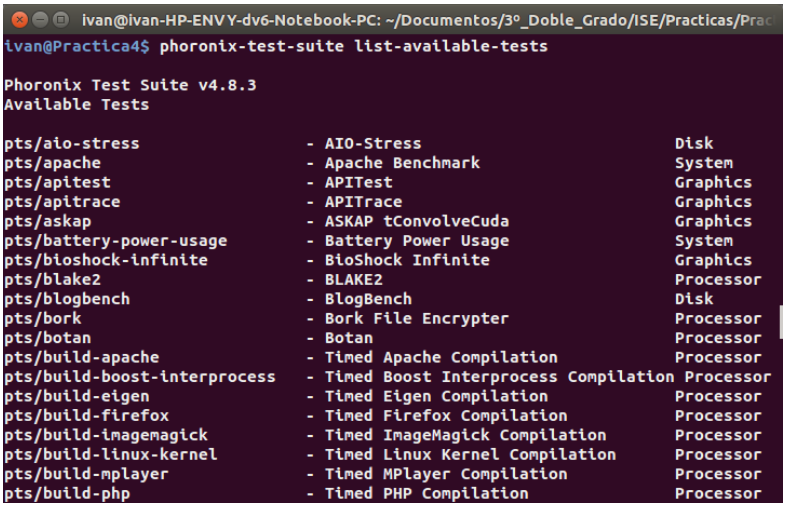

Figura 1.1: Lista de parte de los benchmarks que tiene disponible phoronix

- **Seleccione, instale y ejecute uno, comente los resultados.**
  El benchmark que hemos descargado es el que mide el uso de batería: *battery-power-usage*. Para instalarlo, ejecutamos el comando *install* de phoronix. Para ejecutarlo utilizamos el comando *run*. Al ejecutarlo, se nos dice con que nombre queremos que se guarden los resultados. Tras esto, ejecuta el benchmark seleccionado. En nuestro caso, el benchmark mide el uso de la batería sin hacer nada pero con la pantalla encendida, con la pantalla apagada y reproduciendo un video, por lo que intuimos que está enfocado a ver cuanto usa la GPU la batería. Tras la ejecución del mismo, nos preguntan que si queremos ver información del benchmark, en la que se incluye la gráfica que muestra el uso que ha hecho nuestro computador durante el mismo:

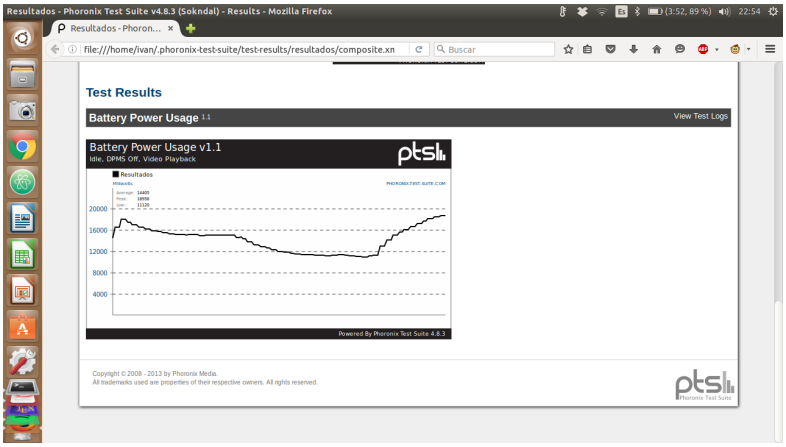

Figura 1.2: Gráfica que muestra el uso de batería durante el benchmark en MiliWatios.

La primera parte del gráfico hace referencia a la parte en la que el computador tenía la pantalla encendida sin hacer nada. Hay un pico al iniciarse esta etapa pero supongo que será por la iniciación del test. Después el uso de batería decrece, lo que quiere decir que corresponde a la parte del benchmark en la que la pantalla permanecía apagada. Tras esto, crece abruptamente, lo que quiere decir que comienza la parte en la que se está reproduciendo un vídeo.

Para obtener la información que recoge el benchmark directamente podemos ejecutar los distintos comandos *result-file-to-...* En este caso, ejecutaremos el que nos lo muestro en texto. Esta es la salida:

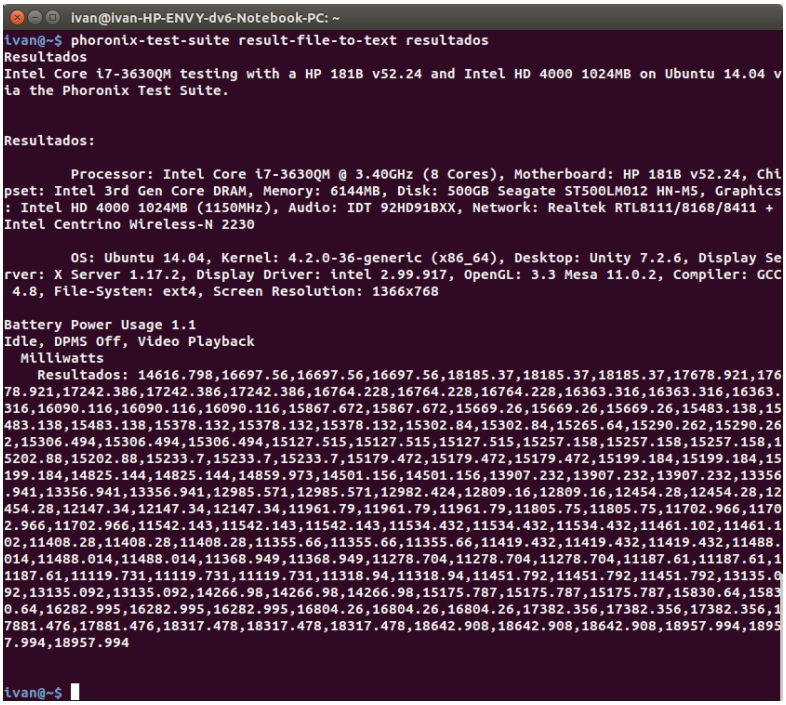

Figura 1.3: Información sobre el test extendida.

1.2. Tests de estrés para Webs.

1.2.1. Apache Benchmarck

En este apartado vamos a responder las preguntas relacionadas con Apache Benchmark(comando ab).

- **De los parámetros que le podemos pasar al comando ¿Qué significa -c 5 ? ¿y -n 100? Monitorice la ejecución de ab contra alguna máquina (cualquiera) ¿Cuántos procesos o hebras crea ab en el cliente?**
  Según la documentación oficial de Apache(la reference al comando ab) [3], la opción *-n request* fija el número de peticiones que se harán al servidor de apache a *request*. La opción *-c concurrency*, a su vez, fija el número de peticiones concurrentes que dejaremos hacer al cliente.

  Tras esta explicación, vayamos ahora a monitorizar un equipo. Para ello, desde la máquina cliente ejecutamos el siguiente comando para bombardear a peticiones nuestro servidor:

  ab -n 1000 -c 10 http://<server IP>/

el cual envía 1000 peticiones a través de 10 hebras paralelas(peticiones concurrentes). Este es el resultado de nuestro Benchmarking:

```
Server Hostname:        10.0.2.6
Server Port:            80

Document Path:          /
Document Length:        11510 bytes

Concurrency Level:      10
Time taken for tests:   0.689 seconds
Complete requests:      1000
Failed requests:        0
Total transferred:      11783000 bytes
HTML transferred:       11510000 bytes
Requests per second:    1452.12 [#/sec] (mean)
Time per request:       6.886 [ms] (mean)
Time per request:       0.689 [ms] (mean, across all concurrent requests)
Transfer rate:          16709.36 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   1.0      1       8
Processing:     1    6   5.3      5      91
Waiting:        0    5   5.2      4      90
Total:          2    7   5.5      6      93

Percentage of the requests served within a certain time (ms)
  50%      6
  66%      7
  75%      7
  80%      8
  90%     10
  95%     11
  98%     15
  99%     22
 100%     93 (longest request)
```

Figura 1.4: Resultados obtenidos a través del comando ab

Estos resultados nos resumen cuanto ha tardado el servidor en atender a cada petición. A primera vista, nos llama la atención que la diferencia entre el máximo y el mínimo tiempo tardado en dar respuesta a las peticiones es muy grande(90 ms). Si nos fijamos en las últimas filas de la terminal, nos damos cuenta que esto es por la existencia de casos extremos, en los que solo el 1 % de las peticiones ha tardado entre 90 y 22 ms. Todas las demás no han llegado a tardar 22 ms.

Para ver el número de hebras que crea ab en el cliente, vamos a ejecutar *ps -eLf*, que con estas opciones muestra información de las hebras creadas[4], tras mandar las peticiones. Para filtrar la búsqueda del proceso, volcaremos el resultado y lo filtraremos con grep ab. Tras este proceso, vemos la siguiente salida:

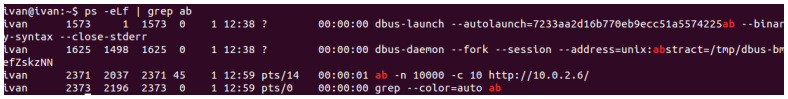

Figura 1.5: Resultado de ejecutar ps -eLf | grep ab en el momento en el que estaba ejecutandose ab

Es claro que la linea que buscamos es la quinta linea.El formato que sigue la salida de ps -eLf es el siguiente: UID(ID del usuario), PID(ID del proceso), PPID(permisos), LWP(ID de la hebra), C (), NLWP(Número de hebras)...(No cito más puesto que solo nos interesa el número de hebras creadas). Así que la columna sexta nos dirá el número de hebras que se crean en el cliente, que en este caso es solo una.

- **Ejecute ab contra las tres máquinas virtuales (desde el SO anfitrión a las máquina virtuales de la red local, en Ubuntu, CentOS y WS) una a una (arrancadas por separado) y muestre y comente las estadísticas. ¿Cuál es la que proporciona mejores resultados? Fíjese en el número de bytes transferidos, ¿es igual para cada máquina?**

Vamos a bombardear a cada máquina con las mismas peticiones y el mismo límite de concurrencia. Vamos a lanzar 1000 peticiones con, como mucho, 10 peticiones a la vez.

La primera máquina a la que vamos a bombardear con peticiones es Windows. Estos son los resultados:

```
Server Software:        Microsoft-IIS/7.0
Server Hostname:        10.0.2.8
Server Port:            80

Document Path:          /
Document Length:        689 bytes

Concurrency Level:      10
Time taken for tests:   0.559 seconds
Complete requests:      1000
Failed requests:        0
Total transferred:      932000 bytes
HTML transferred:       689000 bytes
Requests per second:    1788.80 [#/sec] (mean)
Time per request:       5.590 [ms] (mean)
Time per request:       0.559 [ms] (mean, across all concurrent requests)
Transfer rate:          1628.09 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   1.7      1      14
Processing:     0    4   3.1      3      21
Waiting:        0    3   2.4      3      15
Total:          2    5   3.3      5      22

Percentage of the requests served within a certain time (ms)
  50%      5
  66%      6
  75%      7
  80%      7
  90%     10
  95%     11
  98%     15
  99%     15
 100%     22 (longest request)
ivan@ivan:~$
```

Figura 1.6: Resultados del bombardeo de peticiones a Windows

La siguiente máquina a bombardear es Ubuntu. Estos son los resultados:

```
Server Software:        Apache/2.4.7
Server Hostname:        10.0.2.6
Server Port:            80

Document Path:          /
Document Length:        11510 bytes

Concurrency Level:      10
Time taken for tests:   0.724 seconds
Complete requests:      1000
Failed requests:        0
Total transferred:      11783000 bytes
HTML transferred:       11510000 bytes
Requests per second:    1381.06 [#/sec] (mean)
Time per request:       7.241 [ms] (mean)
Time per request:       0.724 [ms] (mean, across all concurrent requests)
Transfer rate:          15891.64 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   1.7      1      22
Processing:     1    6   4.1      5      33
Waiting:        0    4   3.0      4      21
Total:          2    7   4.3      6      34

Percentage of the requests served within a certain time (ms)
  50%      6
  66%      7
  75%      8
  80%      8
  90%     12
  95%     16
  98%     20
  99%     24
 100%     34 (longest request)
```

Figura 1.7: Resultados del bombardeo de peticiones a Ubuntu

La última máquina a bombardear es CentOs. Estos son los resultados:

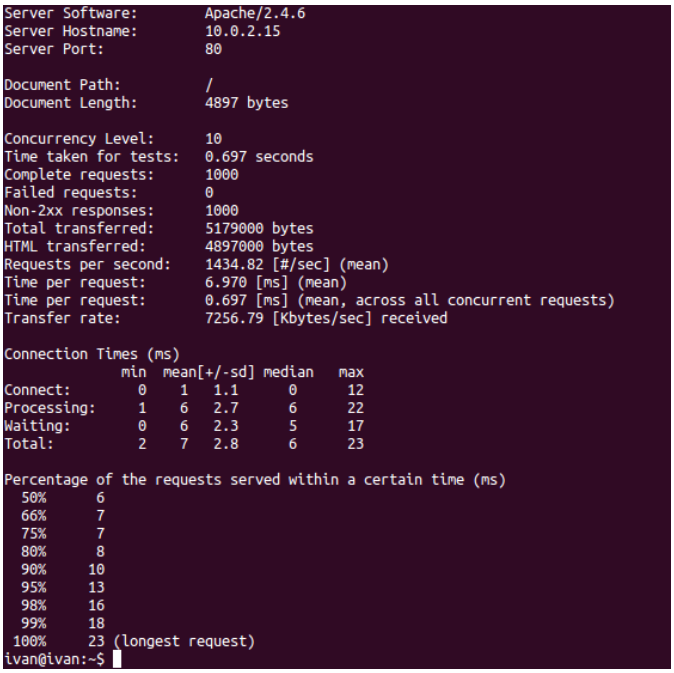

Figura 1.8: Resultados del bombardeo de peticiones a CentOS

En cada informe nos muestra la cantidad de información que se envía. En este caso, Ubuntu manda mucha información(11.510B de información), Windows envía sólo 689B de información y CentOs, 4.897B. Esto desencadena en que, obviamente, la velocidad que tardará Ubuntu en servir la página será mayor que las otras dos distribuciones, ya que la página en cuestión es más pesada.
Como venimos anunciando, Ubuntu ha tardado más en servir las 1000 peticiones(0.724 seg), seguido muy de cerca, sorprendentemente, por Windows(0.559 seg). CentOS ha tardado 0.697 seg.

Por tanto, la relación entre bytes servidos y tiempo de media en servirlos quedaría así:

Máquina	Bytes/ms
Windows	137
Ubuntu	1644
CentOS	699

Si nos fijásemos solo en estos resultados, parece que lo que ha tardado Ubuntu se debe única y exclusivamente a la cantidad de información que manda. Sin embargo,

ya que aparte de mandar información también se realizan otras operaciones que no están tenidas en cuenta en el tiempo medio de respuesta, no podemos concluir que ninguna máquina se comporte mejor o peor realmente.

1.2.2. Gatling

- **¿Qué es Scala? Instale Gatling y pruebe los escenarios por defecto.**
  Scala [5] es un lenguaje de programación que usa los conceptos de orientación a objetos y los conceptos funcionales, formando un lenguaje multiparadigma. Está diseñado para expresar patrones de programación y es altamente tipado.

  Tras esta explicación, veamos ahora cómo usar la herramienta gatling, que usa Scala para crear sus tests, para estresar el sistema. Según la guia de inicio rápido suministrada por el proyecto gatling[6], para instalar el programa simplemente hay que descomprimir un archivo que te descargas de la misma página para tener el programa en funcionamiento. Sin embargo, nosotros vamos a utilizar un paquete que se nos suministra, citado en [12], que contiene las mismas funcionalidades. En el directorio *./bin* se encuentran los ejecutables de la aplicación. Para correr la aplicación, ejecutamos *./bin/gatling.sh*(en Linux) y nos saldrá una muestra de los tests de ejemplo. El que nosotros ejecutaremos será el test básico. Al terminar, nos dirá que para ver los resultados abramos un link creado en el directorio *./results*. Una de las tablas que crea gatling es la siguiente, donde muestra lo que tarda el sistema en responder a una query de la base de datos. Estos son los distintos datos que recogen:

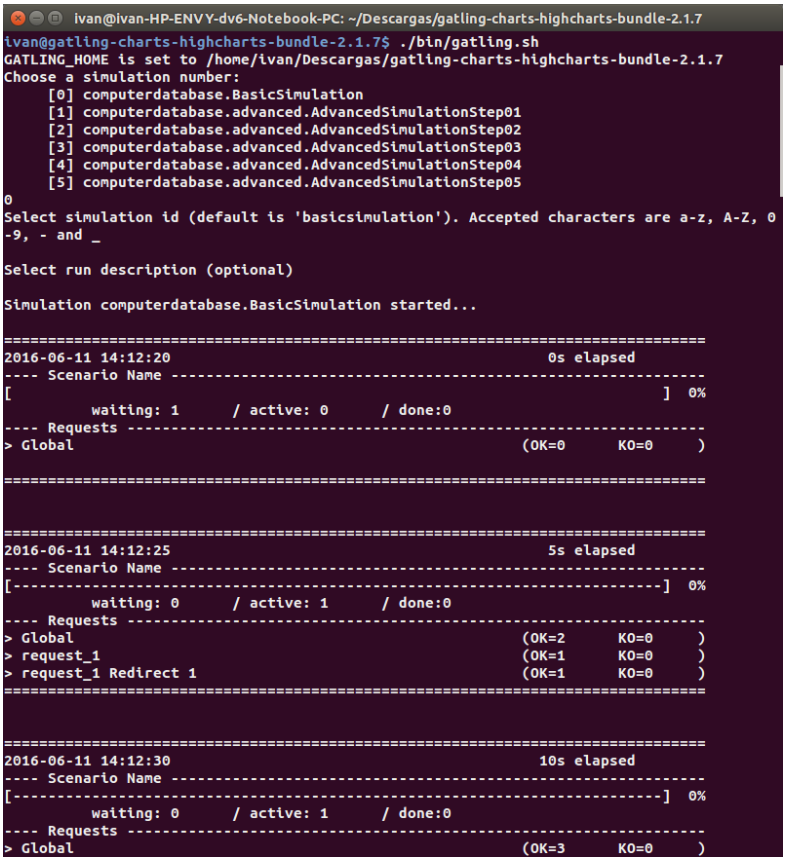

Figura 1.9: Ejemplo de ejecución del tester básico

Para comenzar, cada cierto tiempo nos muestra información de como está transcurriendo el test, con datos como el número de respuestas bien recibidas(OK). Al finalizar, se nos mostrará por pantalla el resultado completo, como en la siguiente imagen:

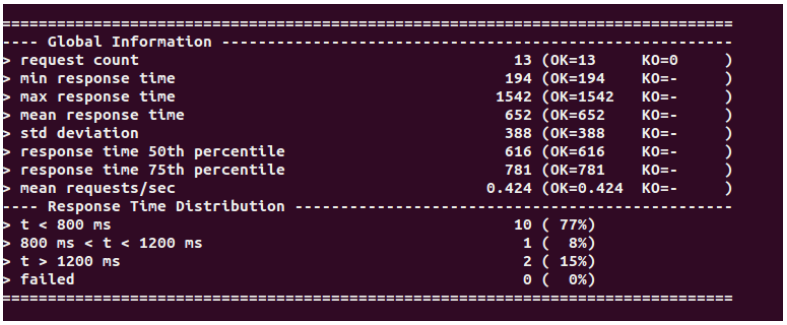

Figura 1.10: Información que saca Gatling por pantalla tras su ejecución.

en la que se nos informa de datos relevantes como algunos percentiles relevantes y los tiempos mínimo y máximo de respuesta. También reparten las peticiones en intervalos de tiempo. Si queremos ver gráficamente estos datos, tendremos que abrir el archivo *index.html* alojado en un directorio creado para nuestro test en *./results*. Los gráficos que allí se muestran son los siguientes entre otros:

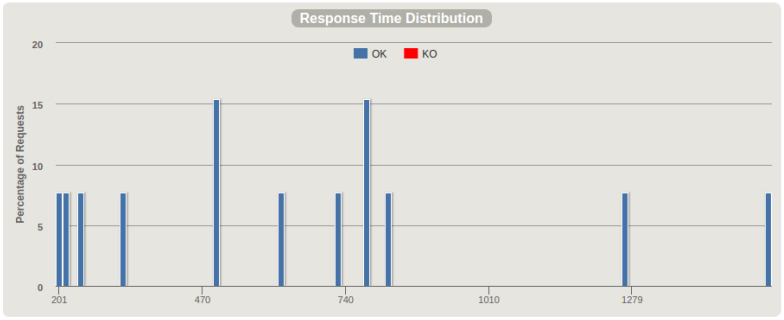

Figura 1.11: Distribución seguida por los tiempos de respuesta de las peticiones

En éste se muestra la distribución de las respuestas en el tiempo de la prueba. Las dos barras más largas muestran que en esos momentos se respondieron el doble de respuestas que en las otras, que sabiendo que solo se realizaron 13 peticiones, son dos respuestas.

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	13	13	0	0%	0.424	194	616	781	1376	1508	1542	652	388
request_...direct 1	1	1	0	0%	0.033	208	208	208	208	208	208	208	0
request_1	1	1	0	0%	0.033	324	324	324	324	324	324	324	0
request_2	1	1	0	0%	0.033	1266	1266	1266	1266	1266	1266	1266	0
request_3	1	1	0	0%	0.033	194	194	194	194	194	194	194	0
request_4	1	1	0	0%	0.033	503	503	503	503	503	503	503	0
request_...direct 1	1	1	0	0%	0.033	725	725	725	725	725	725	725	0
request_5	1	1	0	0%	0.033	494	494	494	494	494	494	494	0
request_6	1	1	0	0%	0.033	816	816	816	816	816	816	816	0
request_7	1	1	0	0%	0.033	774	774	774	774	774	774	774	0
request_8	1	1	0	0%	0.033	242	242	242	242	242	242	242	0
request_9	1	1	0	0%	0.033	1542	1542	1542	1542	1542	1542	1542	0
request_10	1	1	0	0%	0.033	781	781	781	781	781	781	781	0
request_...direct 1	1	1	0	0%	0.033	616	616	616	616	616	616	616	0

Figura 1.12: Tabla de resultados del test de estrés donde gatling mide lo que tarda el sistema en responder a una petición.

En esta tabla se muestran los porcentajes, los tiempos y los percentiles de todas las respuestas. Así por ejemplo, vemos que la respuesta que tardó más tiempo fue la novena, y que la desviación típica en ms es de 388ms.

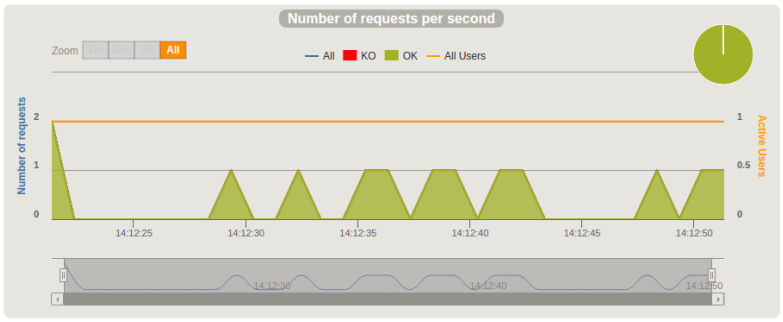

Figura 1.13: Gráfico que muestra el número de peticiones por segundo en el transcurso del test.

En este gráfico se muestra el número de peticiones por segundo y el número de usuarios activos(que en nuestro caso solo seremos nosotros). Sólo al principio se tuvo un promedio de dos peticiones por segundo.

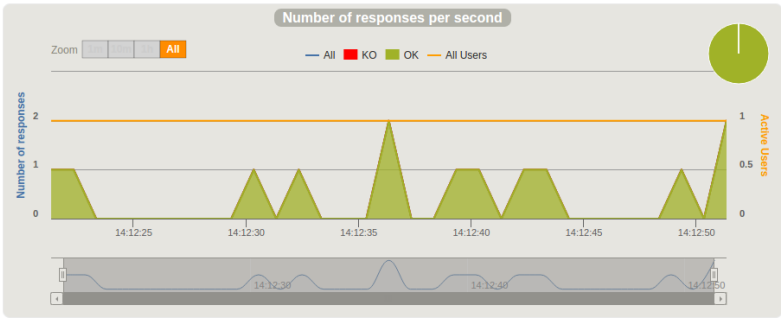

Figura 1.14: Gráfico que muestra el número de respuestas por segundo en el transcurso del test.

Este último gráfico muestra el número de respuestas por segundo y el número de usuarios activos también. Varía del anterior en poca cosa, ya que son las respuestas a las peticiones representadas arriba.

1.2.3. Jmeter

- **Lea el artículo y elabore un breve resumen[7].**
  El artículo pretende poner cara a cara las prestaciones de la herramienta vista anteriormente(Gatling) y Jmeter, una herramienta parecida. Al inicio del artículo deja muy claro que no quieren poner una por encima de la otra, ya que no es la intención de la página(flood.io), que apoya a ambas, eliminar una en función de la otra. Esto es porque piensan que estas herramientas necesitan distintos requerimientos.

Los primeros párrafos describen en que circunstancias se van a llevar a cabo las pruebas con ambas herramientas y por qué. Utilizan, por ejemplo, un servidor con 4 CPUs virtuales en el que han habilitado 15 GB de RAM para asegurarse de que no se producen cuellos de botella en el servidor. También se dejan claros otros detalles, como que se correrá sobre la máquina virtual de Java con una serie de opciones de la misma(aquí no especificadas).

La rutina que seguirá el benchmark consistirá en distintas transacciones hechas por üsuarios", cada una de ellas con distintos requerimientos usando recursos mas o menos lentos para tener una idea de todos los tipos de peticiones que se pueden hacer.

Los últimos parámetros que se tienen en cuenta son la cantidad de peticiones, el nivel de concurrencia y el tiempo del mismo benchmark. Se correrán 10.000 hebras usuario, 30.000 peticiones por minuto y se correrá durante 20 minutos, con un tiempo de descanso de la máquina de 10 minutos.

En este test, puesto que hay varias versiones de Jmeter, se ha pasado a medir las prestaciones de una y otra versión. Nosotros solo evaluaremos la diferencia que hay entre las dos herramientas(Gatling y Jmeter) no entre las distintas versiones de Jmeter.

A continuación, pasamos a describir los resultados:

- Para empezar, la media que calculaba cada uno de los tests era muy parecida, al igual que la desviación típica, por tanto podemos coincidir en que se obtienen casi los mismos resultados con ambos testers.
- En cuanto a uso de red, cabe notar que Gatling no guarda información sobre el tamaño de la respuesta. Sin embargo, flood.io usa una estimación basándose en las cabeceras. Aun así, la estimación es optimista. El siguiente, es un gráfico del uso de cada herramienta de la red:

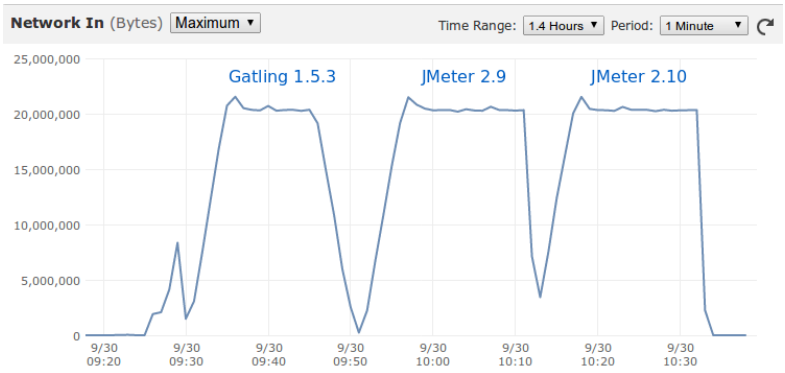

Figura 1.15: Uso de red de Gatling y las dos versiones de Jmeter.

- Jmeter usa más recursos en la máquina virtual de Java.
- Jmeter usa más la CPU y más memoria. El siguiente gráfico muestra el porcentaje de utilización de la CPU:

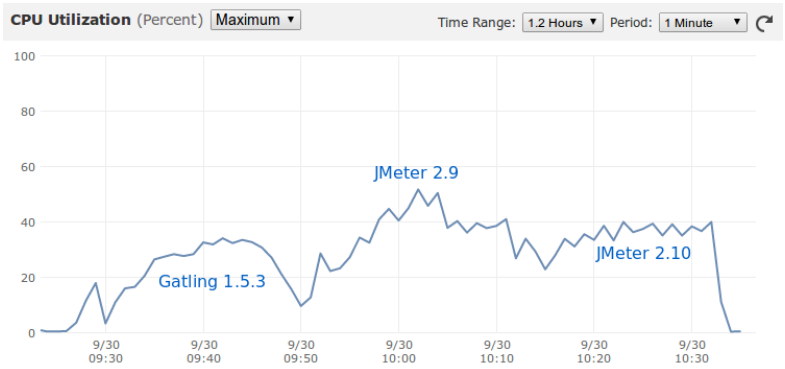

Figura 1.16: Uso de la CPU de Gatling y ambas versiones de Jmeter

- Ambas herramientas tienen tiempos de respuesta muy parecidos(desviación típica muy pequeña), lo cual, según el artículo, este parámetro no tiene por qué tenerse en cuenta en los tiempos
- Ambas herramientas se mantienen estables en el momento en el que reciben 30.000 peticiones por minuto.
- Ambas herramientas pueden aguantar a 10.000 usuarios concurrentemente, lo cual se suele considerar una cantidad agresiva.

Resumiendo, ambas herramientas son muy parecidas, salvo en pequeñas diferencias, como son la medición de uso de red(que Gatling no hace, pero es fácil de medir con otro tester) y el uso de CPU(Jmeter usa ligeramente más). La conclusión es

que el uso de una u otra herramienta termina siendo una cuestión subjetiva que dependerá del usuario.

- **Instale y siga el tutorial en [8] realizando capturas de pantalla y comentándolas. En vez de usar la web de jmeter, haga el experimento usando alguna de sus máquinas virtuales (Puede hacer una página sencilla, usar las páginas de phpmyadmin, instalar un CMS, etc.).**

  Para empezar, vamos a instalar Jmeter. Para ello, simplemente tendremos que descargar el archivo comprimido que se descarga de la misma página [8] y descomprimirlo. Tras verificar que tenemos los requerimientos necesarios [9], vamos al manual para crear test para páginas web [10].

  Nuestro test consistirá en crear cinco usuarios que mandarán peticiones a páginas web. Para poder crear estos test de páginas web vamos a necesitar cuatro elementos principales: Thread Group, HTTP Request, HTTP Request Defaults, y Graph Results.

  - Para el primer elemento(Thread Group) vamos a ir a la interfaz para crear un grupo de hilos:

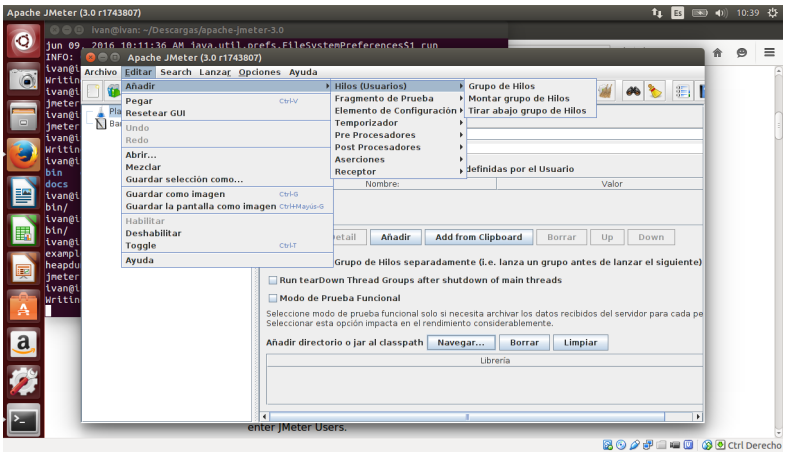

Figura 1.17: Pestaña donde se encuentra la interfaz para crear el grupo de hilos.

Luego, en la interfaz, vamos a modificar algunos valores por defecto. El número de hilos lo modificamos a 5. Además, el numero de repeticiones que vamos a exigir es de 2. Así debería quedar:

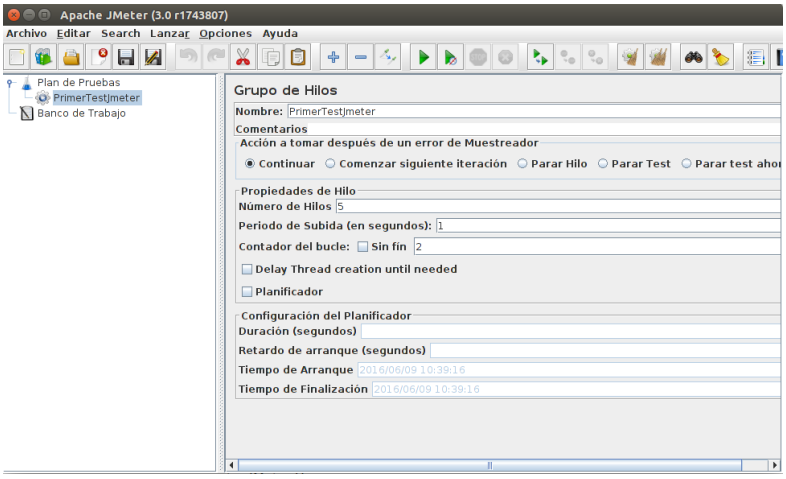

Figura 1.18: Configuración de Jmeter deseada.

- Ahora vamos a crear los valores por defecto de las peticiones que se harán. Para ello, clicamos con el botón derecho sobre el grupo de hilos creado ->Configuración por defecto ->Valores por defecto para configuración HTTP. Nosotros solo modificaremos el campo IP y el puerto, a la máquina que queramos testear. En nuestro caso, a la máquina con IP 10.0.2.15, al puerto 80. Aquí se ven los cambios realizados:

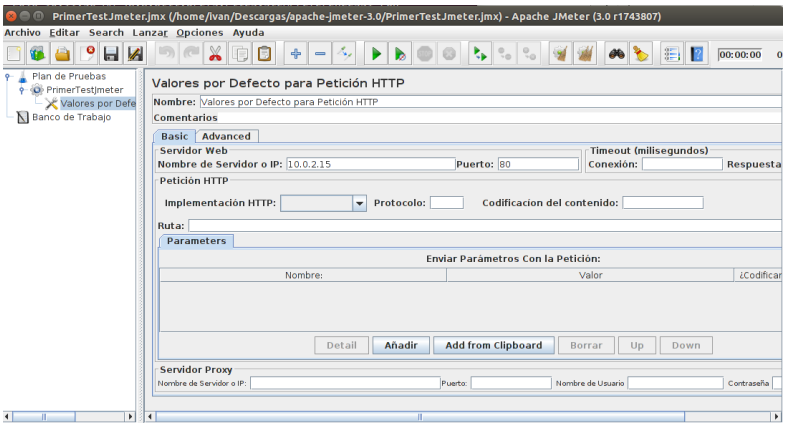

Figura 1.19: Valores por defecto modificados en Jmeter

Jmeter también nos da la opción de incluir Cookies, pero como nuestra página no las usa, nos saltaremos este paso de la guía.

- Con el objeto creado anteriormente solo hemos modificado los valores por defecto, pero no habremos mandado ninguna petición. Ahora si vamos a crear la

propia petición. Para ello, clicamos con el botón derecho sobre nuestro grupo de hilos y le damos a Añadir->Muestreador->Petición HTTP. En ella lo único que necesitaremos cambiar es el nombre, si queremos, y la ruta. No necesitamos especificar la ruta del servidor, ya que ya la tenemos configurada en las opciones por defecto. Así quedaría la petición HTTP:

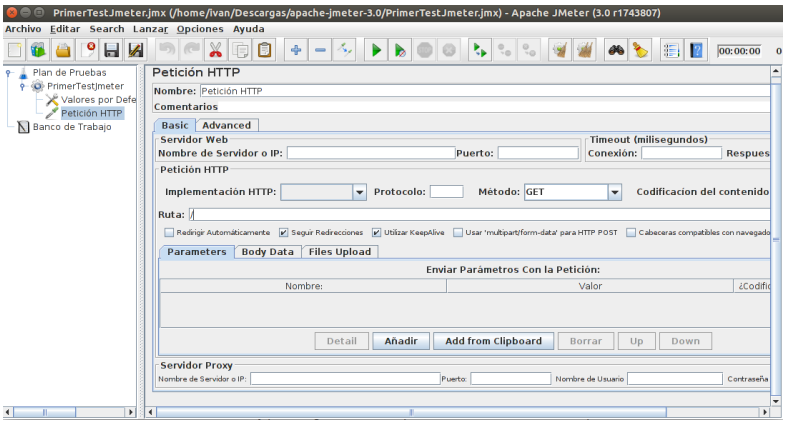

Figura 1.20: Valores modificados de la petición HTTP.

- Por último, tenemos que añadir un elemento que escuche. Este elemento será el responsable de mostrar los resultados de la ejecución de Jmeter, que en nuestro caso será un gráfico. Para ello, clicamos de nuevo enel grupo de hilos y vamos a Añadir->Receptor->Grafico de Resultados. Tras esto, debemos especificar el nombre del archivo al que se volcarán los resultados. Esta es la interfaz que ofrece:

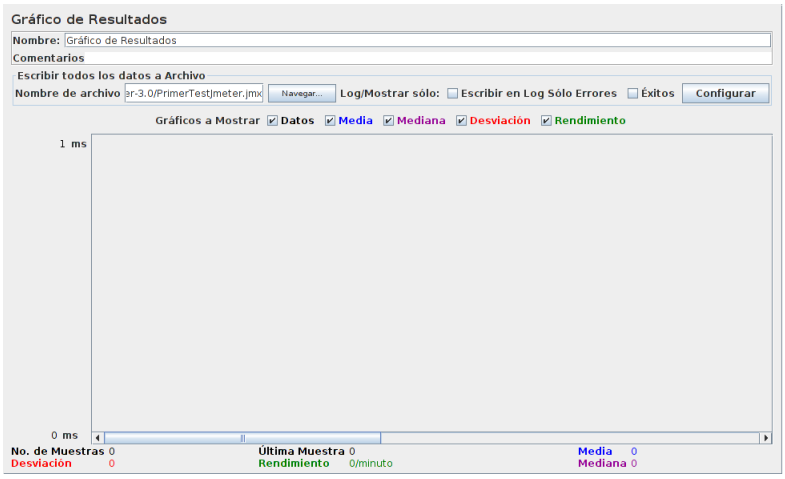

Figura 1.21: Interfaz que ofrece Jmeter para creación de gráficos.

Tras esto, ejecutamos varias veces el test y este es el gráfico que nos queda:

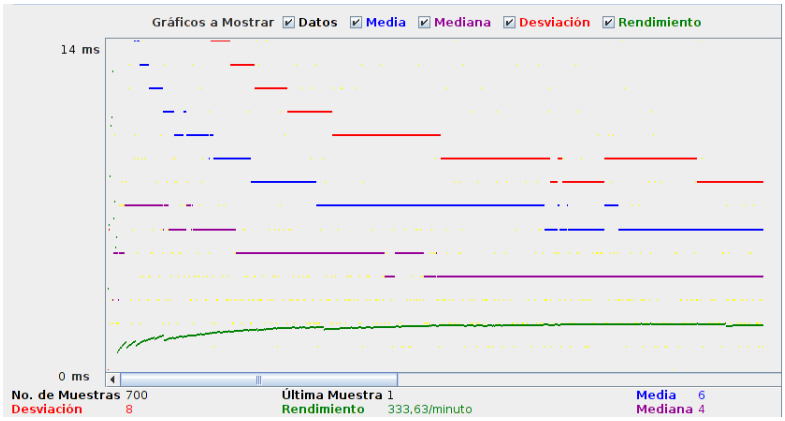

Figura 1.22: Resultados de la ejecución de Jmeter

En la parte de abajo del gráfico, vemos valores clave del mismo gráfico, entre ellos la media y la mediana. Según estos, la mitad de las peticiones, que han sido 700, se han llevado a cabo en un tiempo menor a 4 ms, con una media total de 6ms. También vemos en la gráfica que las primeras peticiones tardaban más, pero que al final el tiempo de respuesta y todos los parámetros que en el intervienen se estabilizan.

1.3. Benchmark

En esta última parte de la memoria, tenemos que programar un benchmark e incluir el objetivo del mismo, que unidades, variables o puntuaciones utilizará, instrucciones de uso y un ejemplo en el que se analizen los resultados.

Para la realización del benchmark se ha consultado el benchmark publicado el año pasado por el compañero de clase Oscar [11] y se ha modificado. A continuación se explican los campos que se requieren:

- Para comenzar, el Benchmark pretende medir los MFLOPS del computador en el que se ejecuten. Para ello, la rutina que ejecutará será la de invertir una matriz de dimensión 1000 aleatoria por el método de descomposición LU y se calcularán las operaciones en coma flotante necesarias para ella. Se medirá el tiempo necesario en la inversión y, a partir de los datos de tiempo y número de operaciones, se calcularán los MFLOPS.
- Para que el benchmark sea más homogéneo, este calculo se hará 5 veces y se calculará la media.
- La forma de ejecución de este Benchmark será simplemente ejecutar el script benchmark.sh, que sacará por pantalla el número de MFLOPS del computador. El funcionamiento de este script es el siguiente: primero compila los programas necesarios para la ejecución(el que invierte la matriz y el que calcula la media de los datos). Tras esto, llama y guarda los resultados de la ejecución del programa que calcula la inversa de la matriz las cinco veces dichas anteriormente. Tras esto, calcula la media de estos valores y lo muestra en pantalla.

Se pasa ahora a analizar los resultados de ejecutar el script en varias computadoras:

Procesador	MFLOPS
Intel® Core™ i7-3630QM CPU @ 2.40GHz x 8	507.047
Intel® Core™ i7-3630QM CPU @ 2.40GHz	438.071
2.4 GHz Intel® Core™ 2 Duo	259.229

Los resultados de las dos primeras pruebas corresponden a mi máquina host con SO Ubuntu 14.04, y a una máquina virtual corriendo encima de esta, con Ubuntu 14.04 también. Es clara la diferencia entre una y otra, ya que a más MFLOPS, mejores prestaciones para el cálculo, lo cual es razonable al comparar una máquina y sus máquinas virtuales. La última medición se realizó en otra máquina con SO MacOS, así que la comparación de prestaciones decidirá que máquina es mejor para el cálculo científico. En este caso, vemos claramente que el tercer computador tiene peores prestaciones(en cuanto a operaciones en coma flotante se refiere) ya que,

interpretando los datos, ejecuta menos operaciones en coma flotante por segundo que el primero.

Se adjunta el código del benchmark en el archivo .zip de la entega.

Referencias

[1] http://www.phoronix-test-suite.com/?k=downloads

[2] http://www.phoronix-test-suite.com/documentation/phoronix-test-suite.pdf

[3] https://httpd.apache.org/docs/2.4/programs/ab.html

[4] http://linux.die.net/man/1/ps

[5] http://www.scala-lang.org/

[6] http://gatling.io/docs/2.2.1/quickstart.html

[7] https://blog.flood.io/benchmarking-jmeter-and-gatling/

[8] http://jmeter.apache.org/download_jmeter.cgi

[9] http://jmeter.apache.org/usermanual/get-started.html

[10] http://jmeter.apache.org/usermanual/build-web-test-plan.html

[11] https://github.com/oxcar103/Benchmark-103

[12] https://repo1.maven.org/maven2/io/gatling/highcharts/gatling-charts-highcharts-bundle/2.1.7/gatling-charts-highcharts-bundle-2.1.7-bundle.zip