

ANEXO I :

TEST DE

CARGA

1. HERRAMIENTAS UTILIZADAS.

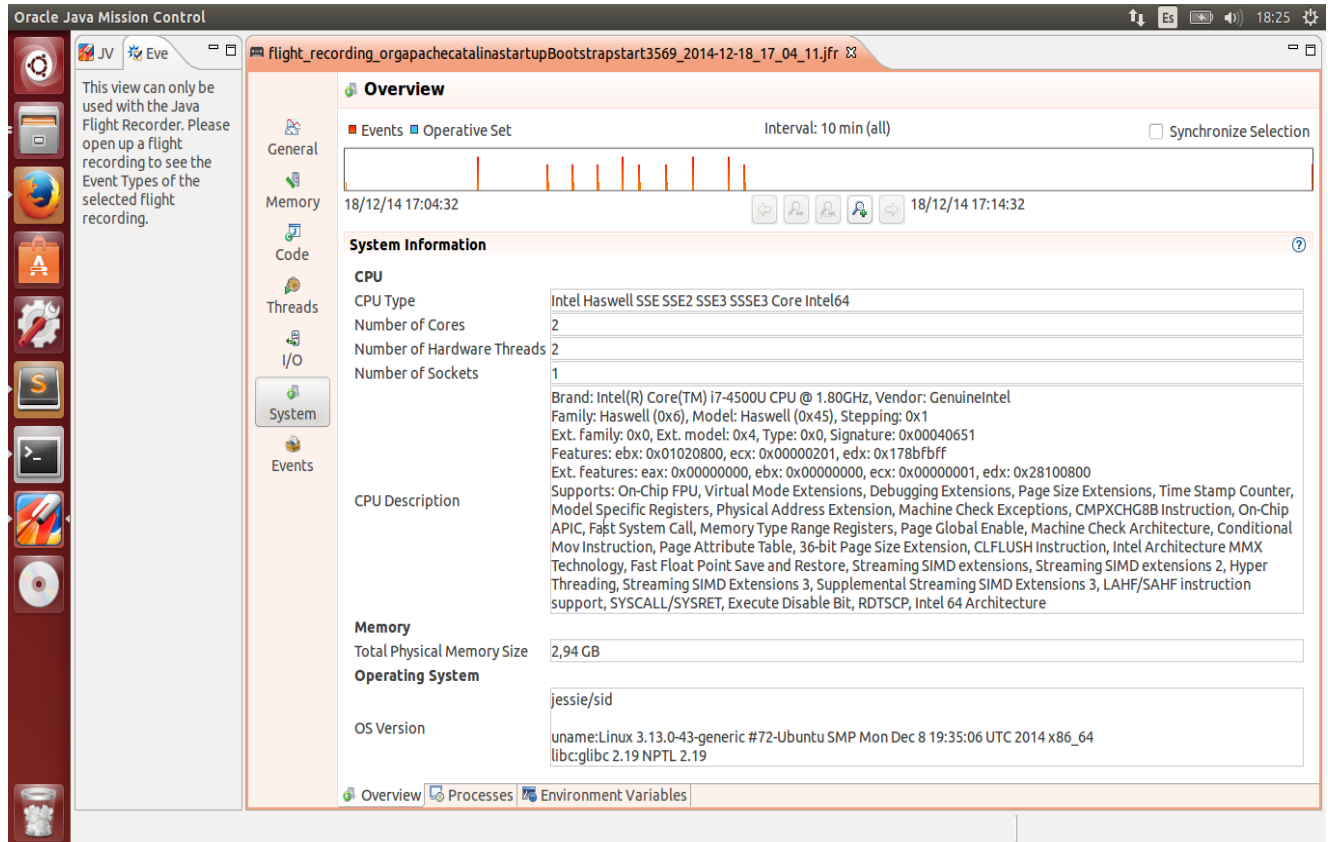
Como se ha comentado previamente en este informe, para la ejecución de las pruebas de carga se ha desarrollado un servicio REST de búsqueda de tuits sobre el que se realizarán dichas pruebas. Se han utilizado las siguientes herramientas:

- **Mockaroo:** Generador de “mock data” utilizado para poblar nuestra base de datos MongoDB con 1000 tuits aleatorios.
- **Spring Framework:** Utilizado para la implementación del servicio REST a través del bean “@RestController”. Básicamente, su funcionamiento consiste en responder a una URL con el parámetro “keywords”, que serán las palabras clave que utilizará para realizar la búsqueda de tuits en base de datos, y devolverlos en formato JSON.
- **ab (Apache Bench Tool):** Herramienta de “benchmarking” para un servidor HTTP y que permite la emisión de muchas peticiones concurrentes. Utilizada para los tests de carga simulando un gran número de peticiones de forma gradual, dejando intervalos en medio de descanso.
- **Java Mission Control:** Herramienta para la recolección continua de información de bajo nivel acerca del sistema en ejecución, más concretamente a través de “flight recordings” que nos permiten recolectar información sobre el Tomcat ejecutando el servicio REST durante un período, establecido previamente, de tiempo.

2. CONTEXTO.

La ejecución de las pruebas se ha realizado sobre una máquina virtual VirtualBox, donde son más notables los abusos de recursos, como la memoria.

Especificaciones:



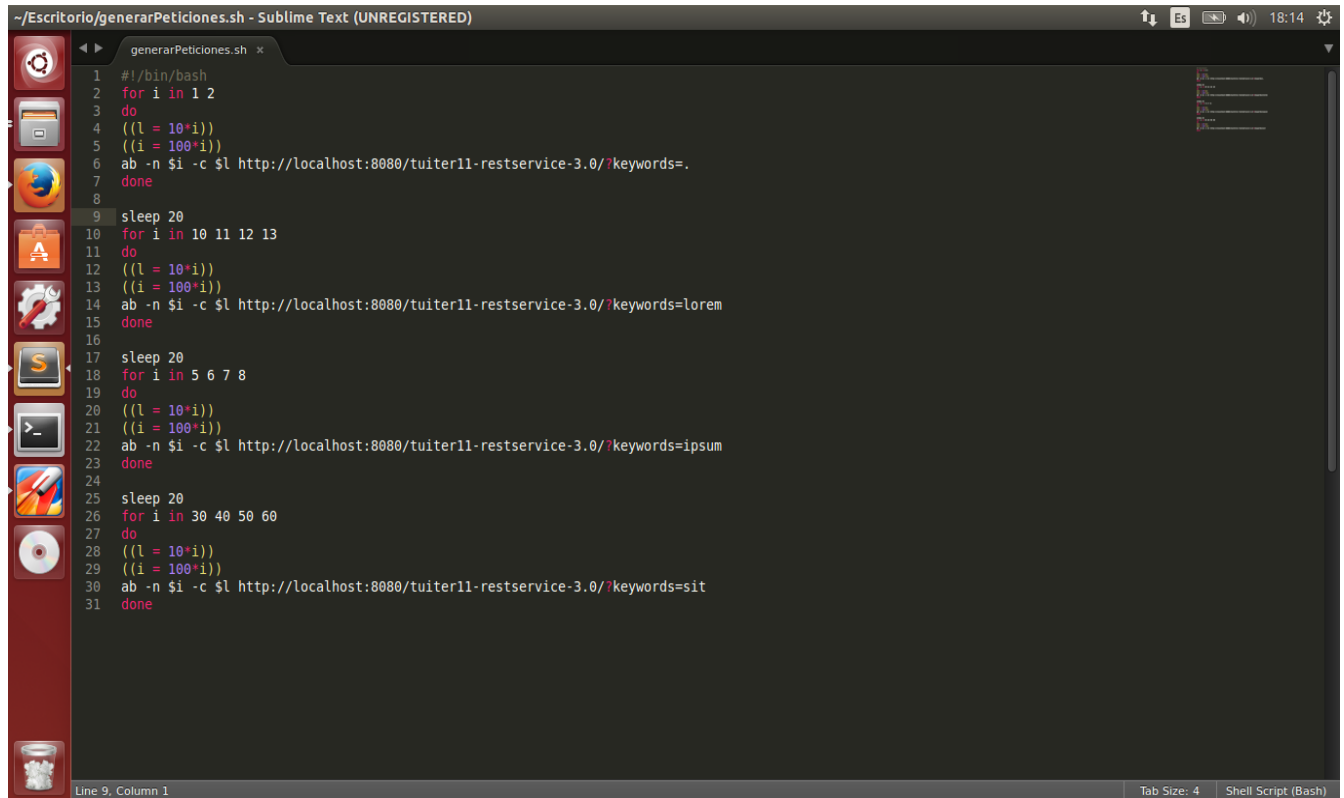
The screenshot displays the Oracle Java Mission Control interface. On the left is a sidebar with icons for General, Memory, Code, Threads, I/O, System, and Events. The main panel is titled 'Overview' and shows a timeline of events. Below the timeline, the 'System Information' section provides details about the CPU, Memory, and Operating System.

System Information

Category	Details
CPU	
CPU Type	Intel Haswell SSE SSE2 SSE3 SSSE3 Core Intel64
Number of Cores	2
Number of Hardware Threads	2
Number of Sockets	1
CPU Description	Brand: Intel(R) Core(TM) i7-4500U CPU @ 1.80GHz, Vendor: GenuineIntel Family: Haswell (0x6), Model: Haswell (0x45), Stepping: 0x1 Ext. Family: 0x0, Ext. model: 0x4, Type: 0x0, Signature: 0x00040651 Features: ebx: 0x01020800, ecx: 0x00000201, edx: 0x178bfbf Ext. Features: eax: 0x00000000, ebx: 0x00000000, ecx: 0x00000001, edx: 0x28100800 Supports: On-Chip FPU, Virtual Mode Extensions, Debugging Extensions, Page Size Extensions, Time Stamp Counter, Model Specific Registers, Physical Address Extension, Machine Check Exceptions, CMPXCHG8B Instruction, On-Chip APIC, Fast System Call, Memory Type Range Registers, Page Global Enable, Machine Check Architecture, Conditional Mov Instruction, Page Attribute Table, 36-bit Page Size Extension, CLFLUSH Instruction, Intel Architecture MMX Technology, Fast Float Point Save and Restore, Streaming SIMD extensions, Streaming SIMD extensions 2, Hyper Threading, Streaming SIMD Extensions 3, Supplemental Streaming SIMD Extensions 3, LAHF/SAHF instruction support, SYSCALL/SYSRET, Execute Disable Bit, RDTSCP, Intel 64 Architecture
Memory	
Total Physical Memory Size	2,94 GB
Operating System	
OS Version	uname:Linux 3.13.0-43-generic #72-Ubuntu SMP Mon Dec 8 19:35:06 UTC 2014 x86_64 libc:glibc 2.19 NPTL 2.19

At the bottom of the main panel, there are tabs for 'Overview', 'Processes', and 'Environment Variables'.

Se ha utilizado un script de Bash para la realización de las peticiones, el cuál es el siguiente:



```
#!/bin/bash
1  for i in 1 2
2  do
3    ((l = 10*i))
4    ((i = 100*i))
5    ab -n $i -c $l http://localhost:8080/twiter11-restservice-3.0/?keywords=.
6  done
7
8
9  sleep 20
10 for i in 10 11 12 13
11 do
12   ((l = 10*i))
13   ((i = 100*i))
14   ab -n $i -c $l http://localhost:8080/twiter11-restservice-3.0/?keywords=lorem
15 done
16
17 sleep 20
18 for i in 5 6 7 8
19 do
20   ((l = 10*i))
21   ((i = 100*i))
22   ab -n $i -c $l http://localhost:8080/twiter11-restservice-3.0/?keywords=ipsum
23 done
24
25 sleep 20
26 for i in 30 40 50 60
27 do
28   ((l = 10*i))
29   ((i = 100*i))
30   ab -n $i -c $l http://localhost:8080/twiter11-restservice-3.0/?keywords=sit
31 done
```

Cabe destacar que el bucle que realiza la primera ráfaga es más corto puesto que realiza peticiones con “keywords=.”, lo que hará que en cada petición se devuelvan las 1000 líneas introducidas previamente en MongoDB con Mockaroo. Luego, después de cada ráfaga con una palabra clave cualquiera, se duerme la ejecución del script durante 20 segundos para dar tiempo a la liberación de recursos, recolector de basura, etc.

3. RESULTADOS.

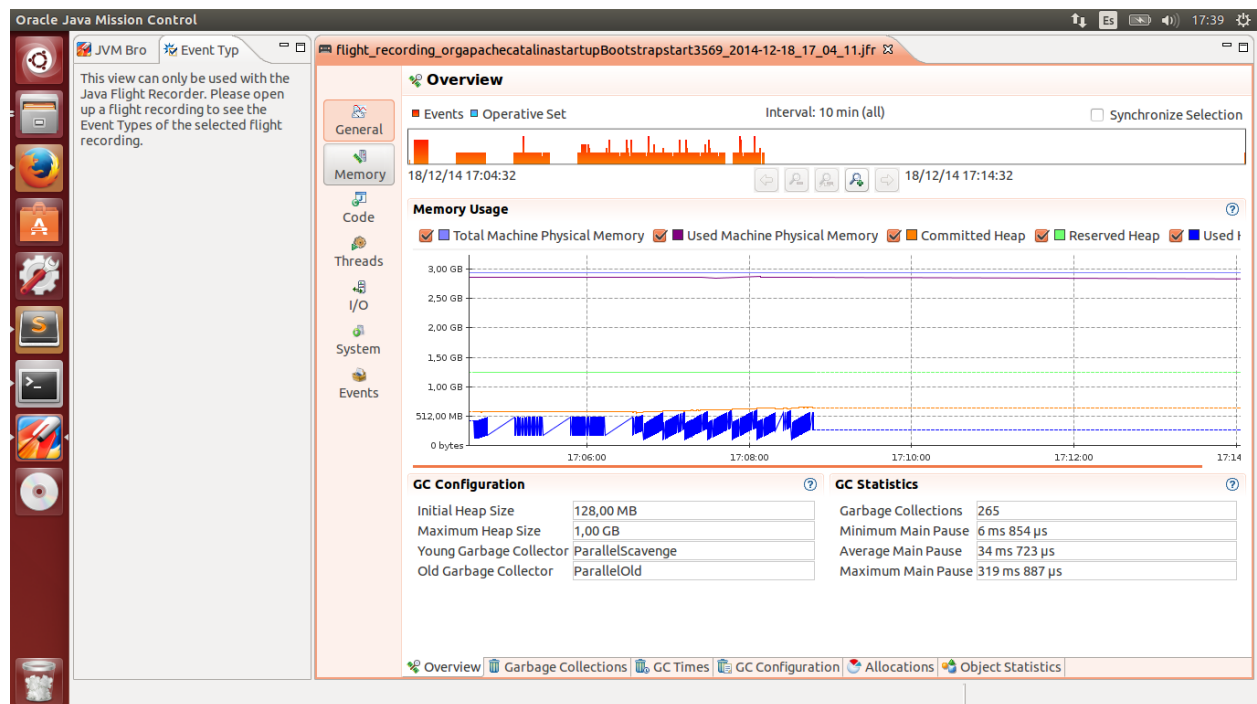
En este apartado mostraremos los resultados obtenidos con los tests de carga:

Uso de Memoria

El Heap es la zona de memoria dinámica, almacena los objetos que se crean, en un principio tiene un tamaño fijo asignado por la JVM (Java Virtual Machine), pero según es necesario se va añadiendo más espacio.

Se puede ver como el Tomcat consume aproximadamente unos 600Mb de un total de 3GB, estando reservados para el Heap 1GB que nunca se llega a consumir, ni se necesita ampliar.

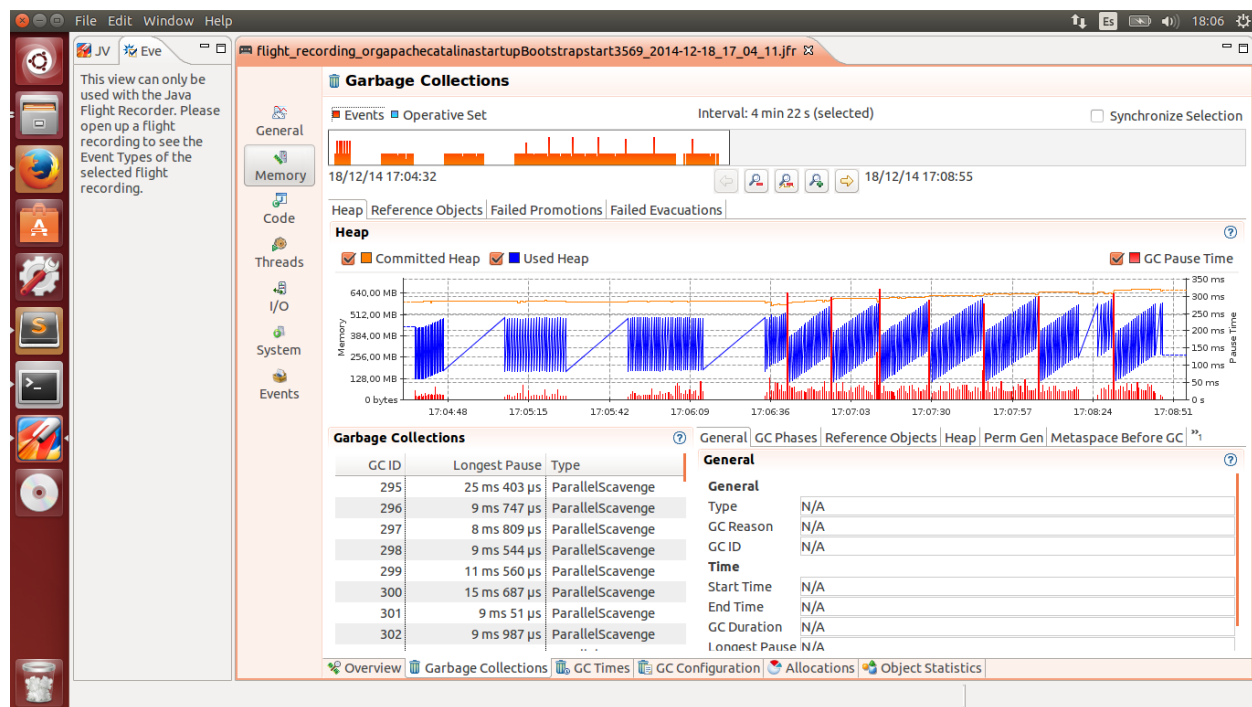
Es interesante ver en esta gráfica como durante los 5 minutos aproximadamente que ha durado el test de carga la memoria consumida por el servidor de aplicaciones no aumenta con el tiempo, lo cual es un primer indicador de la ausencia de memory leaks.



Heap + Garbage Collector

En este otro pantallazo observamos como el recolector de basura interviene para limpiar la memoria ocupada por la aplicación. Los períodos en los que se disparan peticiones son aquellos en donde vemos que la gráfica de memoria empleada por el Heap tiene altibajos más frecuentes. Esto es debido a que con el uso intensivo de memoria, esta se llena más rápidamente, provocando que el Recolector de Basura entre a liberar las partes de la memoria que ya no están siendo apuntadas desde el Stack, lo cual hace que la memoria en uso caiga de nuevo.

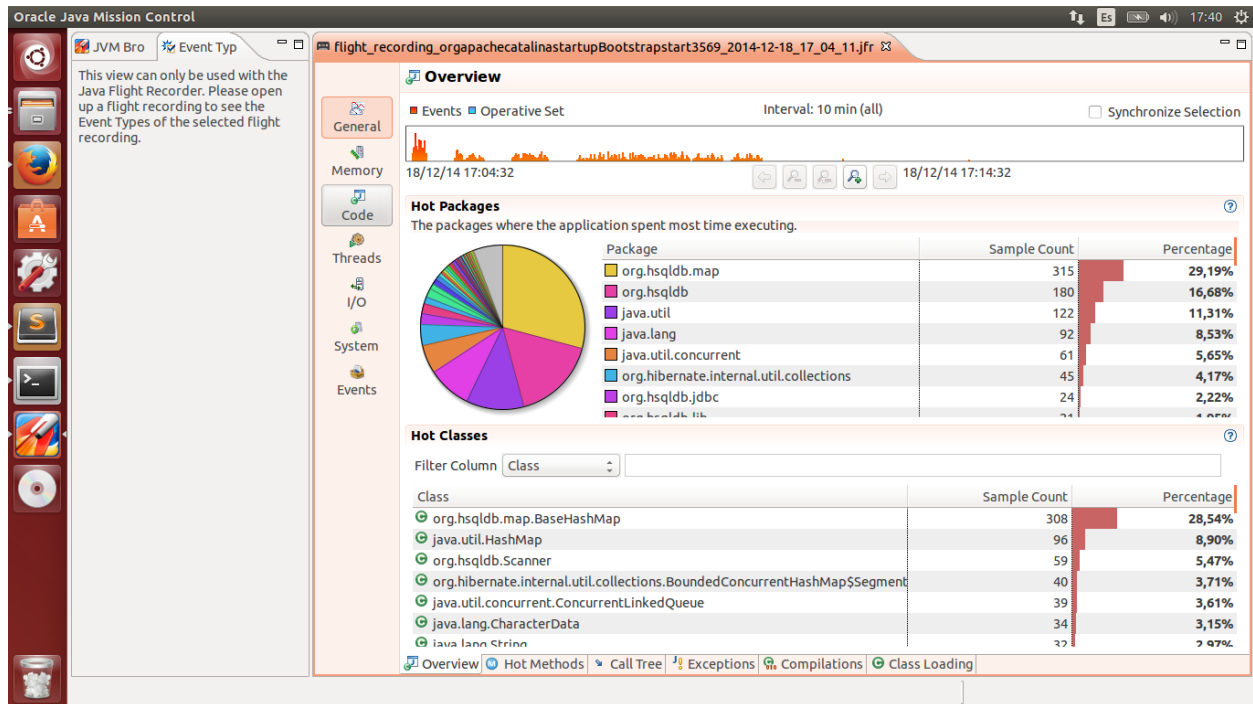
En la primera ráfaga (la que trae más objetos), podemos ver como los picos se suceden con más frecuencia, dado que se usan un número mucho mayor de objetos.



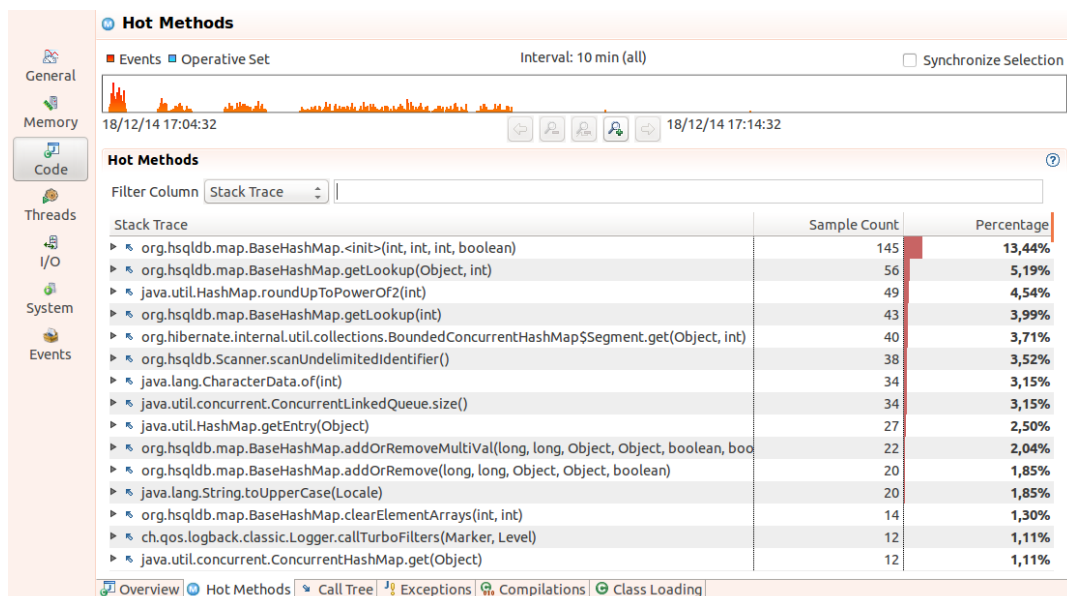
En esta gráfica si se puede observar con más detalle, como cuando en el último tramo, al ejecutar un número más prolongado de peticiones, la memoria después de cada limpieza importante del GC baja siempre hasta el mismo nivel de 128 Mbytes, lo cual nos indica que no hay memoria perdida.

Piezas Problemáticas o Más Usadas

Como se puede apreciar, las clases más empleada son el BaseHashMap de HSQLDB y su interfaz java.util.HashMap, ya que son la clase y interfaz que HSQLDB emplea para realizar su implementación en memoria de las tablas de base de datos, y por tanto todo este tiempo corresponde con tiempo de acceso a base de datos. También se pueden ver como clases más empleadas muchas otras de las empleadas para el acceso a BD como el Scanner de HSQLDB o el driver de jdbc.



Con respecto a los métodos más empleados, estos corresponden también al acceso a BD:



Las clases y métodos de nuestra aplicación apenas son significativas en comparación con los del acceso a Bd como se puede observar en las siguientes imágenes:

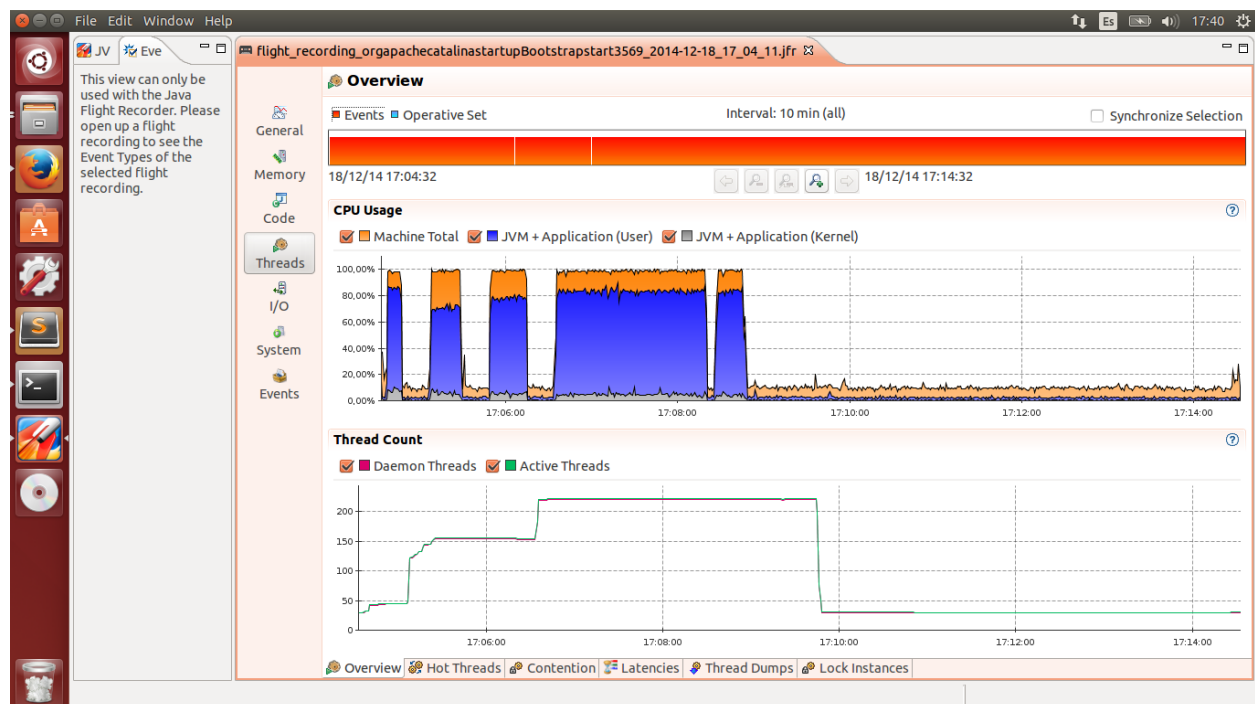
Hot Methods		
Filter Column	Stack Trace	es.udc.fi.dc
Stack Trace		
	Sample Count	Percentage
es.udc.fi.dc.fd.tuit.TuitDtoToTuitConversor.tuitDtosToTuits(List)	2	0,19%
es.udc.fi.dc.fd.account.AccountRepository\$\$FastClassBySpringCGLIB\$\$bfd42224.invoke(int, O	1	0,09%

Hot Classes		
Filter Column	Class	es.udc.fi.dc
Class		
	Sample Count	Percentage
es.udc.fi.dc.fd.tuit.TuitDtoToTuitConversor	2	0,19%
es.udc.fi.dc.fd.account.AccountRepository\$\$FastClassBySpringCGLIB\$\$bfd42224	1	0,09%

Threads Latencies

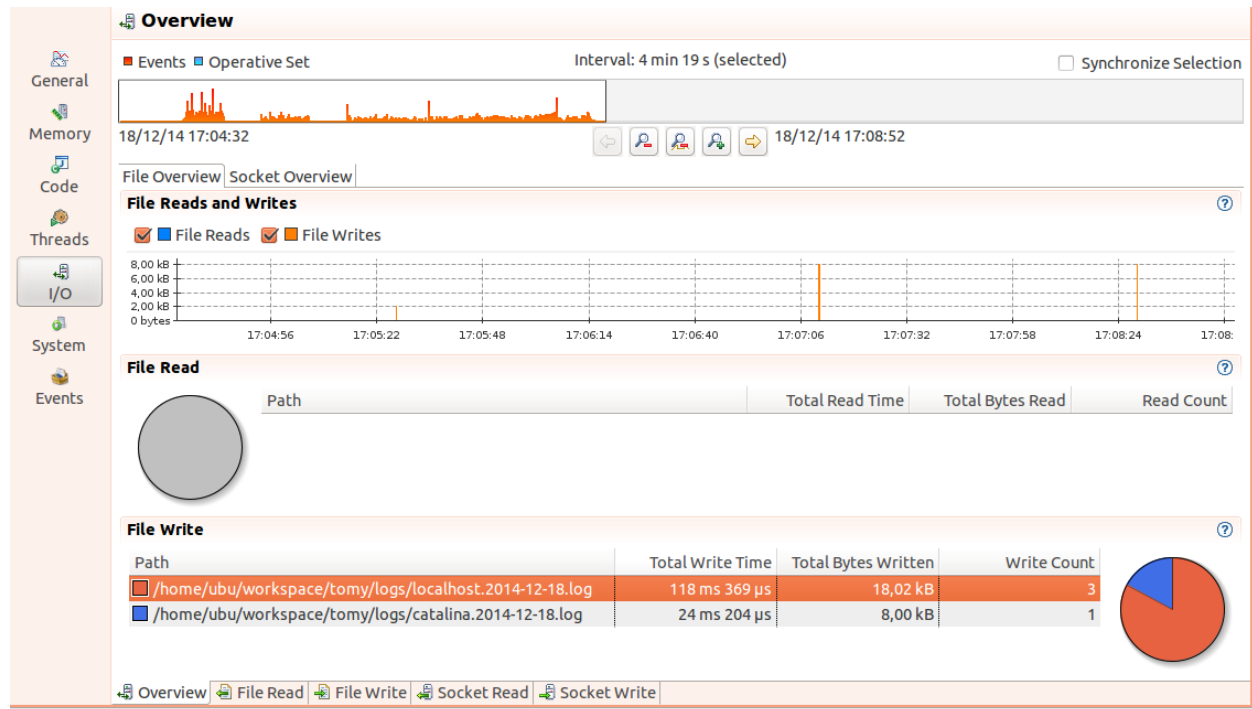
Aquí podemos ver como durante los períodos de máxima actividad, la CPU dedicada a la Máquina Virtual trabaja al 100% siendo de este tiempo, un 80% (aprox.) a atender las peticiones en la JVM de Java.

Por otro lado en la gráfica inferior vemos como el número de threads aumenta en cada ráfaga dado que vamos aumentando progresivamente el número de peticiones concurrentes.

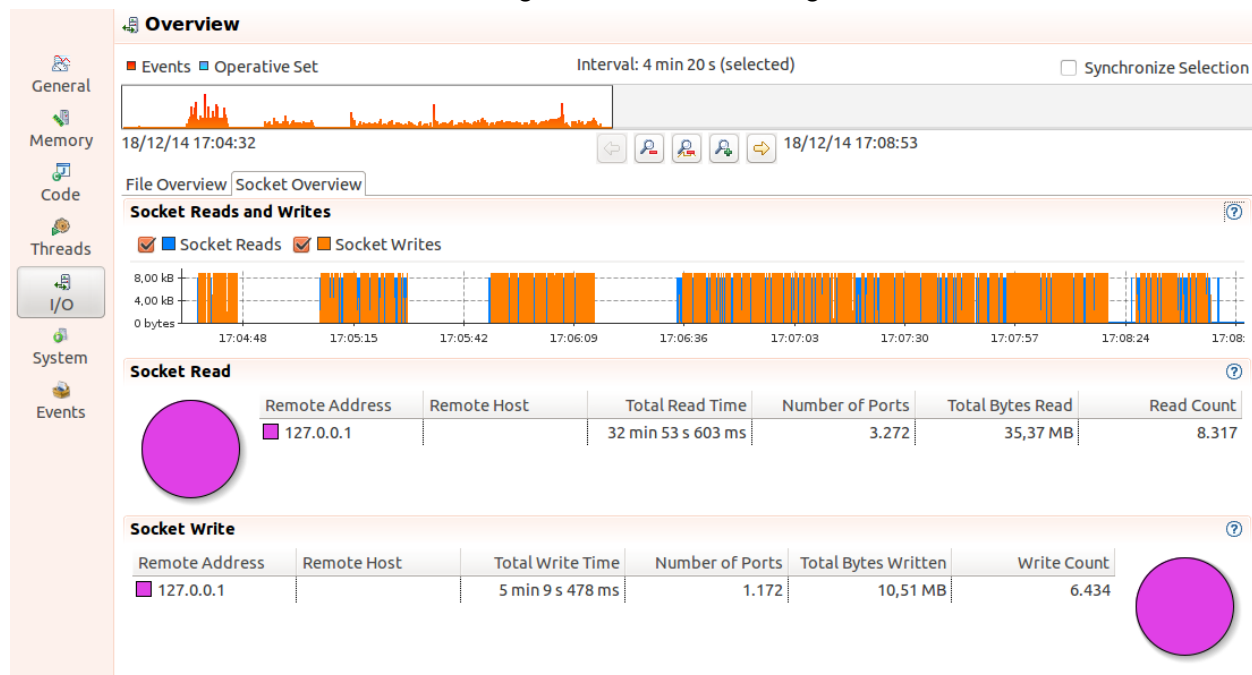


Uso de Sockets y Vista de Entrada/Salida

En cuanto al uso de la Entrada/Salida, a ficheros solo se producen 4 accesos para hacer Log.



Y en cuanto al uso de los Sockets para para atender la peticiones, vemos que todas han sido lanzadas desde localhost, lo cual es lógico en un test de carga.



Allocation Statistics

Aquí vemos como durante la primera rafaga que carga para cada petición todos los tuits que contienen un “.”, es decir, todos los contenidos en la BD, esto provoca una gran creación de objetos por parte de los threads.

