Eusko Jaurlaritzaren Informatika Elkartea
Sociedad Informática del Gobierno Vasco

Guía de uso de HP Diagnostics

Fecha: 20/01/2012

Referencia:

EJIE S.A.
Mediterráneo, 3
Tel. 945 01 73 00*
Fax. 945 01 73 01
01010 Vitoria-Gasteiz
Posta-kutxatila / Apartado: 809
01080 Vitoria-Gasteiz
www.ejie.es

Control de documentación

Título de documento: Arquitectura

Histórico de versiones

Código:
Versión: 1.0
Fecha:
Resumen de cambios:

Cambios producidos desde la última versión

Control de difusión

Responsable: [redacted]	
Aprobado por:	
Firma:	Fecha: 20/01/2012
Distribución:	

Referencias de archivo

Autor: Consultoría y Conocimiento
Nombre archivo: Guía de uso HpDiagnostics.doc

Contenido

1 Introducción.

El presente documento pretende realizar una introducción a los conceptos que derivan del análisis de rendimiento de una aplicación Java, así como proporcionar una visión inicial de la herramienta HP Diagnostics seleccionada en EJIE para tal fin.

Cabe destacar que este documento no pretende ser una guía de usuario de la herramienta, ya que para tal fin existen los manuales de usuario divulgados por el fabricante. Se recomienda su lectura posterior a la de esta guía.

1.1 Introducción al análisis de rendimiento de aplicaciones (profiling)

A lo largo del ciclo de vida del desarrollo de una aplicación nunca se deben de olvidar las consideraciones de rendimiento de la aplicación. Cada pieza de la aplicación debe ser probada para un rendimiento aceptable, teniendo en cuenta los requisitos disponibilidad que dicha aplicación deberá proporcionar cuando esta se encuentre en el entorno de producción.

Un mal rendimiento de la aplicación en un entorno de producción puede provocar un impacto negativo sobre todos los agentes implicados en ella: proveedores, responsables y usuarios finales. Es por este motivo de vital importancia analizar el rendimiento de la aplicación en todas sus fases, haciendo especial hincapié en las fases iniciales del diseño y desarrollo, ya que según se avance en el ciclo de construcción de una aplicación, la detección de un rendimiento pobre del aplicativo tendrá un mayor impacto sobre dichos agentes. Esto se traduce en un incremento de los costes en corregir las anomalías, tal y como se refleja en la figura 1.1:

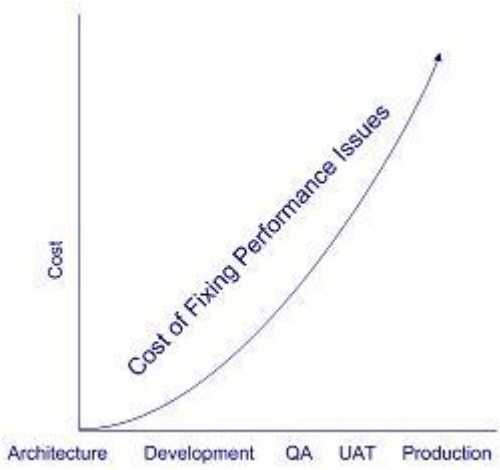


Figura 1.1

Las aplicaciones J2EE son complejas y están construidas sobre una arquitectura de capas (Figura 1.2). Los problemas se pueden suceder en cualquiera de dichas capas, aunque generalmente se producen en la propia aplicación. Aún partiendo del principio de que generalmente los problemas de rendimiento vienen dados por la propia aplicación, en ocasiones puede ser complicado detectar en que punto de las mismas se están produciendo debido a los numerosos componentes que incluye una aplicación: JSPs, servlets, EJBs, Drivers de Base de Datos, servicios web, frameworks, librerías de terceros….

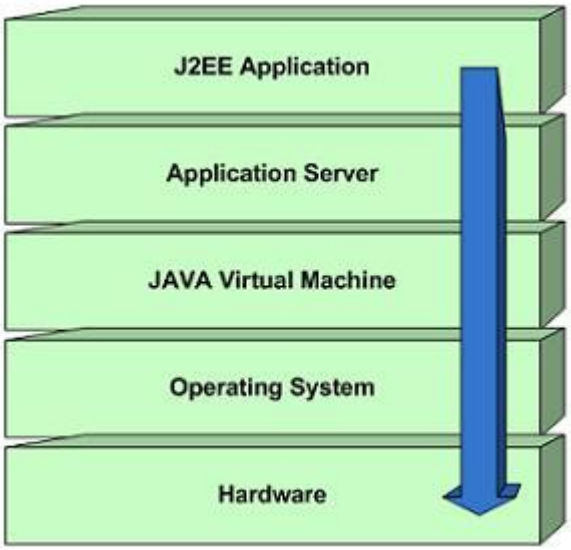


Figura 1.2

Otros factores que caracterizan la complejidad de detectar el punto que está ocasionando problemas de rendimiento, son los parámetros de configuración, ya sean de la propia aplicación, de la configuración del servidor de aplicaciones (threads, pools de conexiones…), de la maquina virtual (Heap) , del sistema operativo (threads, descriptores de ficheros..) o del hardware (RAM, CPU…).

1.1.1. Claves para el análisis del rendimiento de aplicaciones Web

Para una aplicación Web, tres son los principales indicadores mediante los cuales podemos medir su rendimiento:

1. **Tiempo de Respuesta**: Es el indicador mas simple de verificar y a su vez el mas importante. Generalmente el tiempo de repuesta óptimo de una petición se establece entre 1 y 5 segundos.
2. **Utilización de recursos**: La correcta utilización de los recursos del aplicativo impactará notablemente en su rendimiento. Por ejemplo, entendemos como recursos la memoria de la JVM, el pool de conexiones, el pool de threads, tiempos de CPU etc…
3. **Rendimiento de las peticiones**: Entendemos por rendimiento de una petición como la cantidad de trabajo a realizar durante un periodo de tiempo para que ésta pueda ser atendida. Este parámetro también tiene una notable influencia global en la aplicación, ya que un alto rendimiento significa que un único servidor puede procesar un gran número de solicitudes con un impacto mínimo en el tiempo de respuesta. El rendimiento de las peticiones se ve afectado por la eficiencia de las consultas SQL y cómo los componentes se han diseñado en la aplicación.

Podemos identificar algunas de las malas prácticas más comunes que afectan a los indicadores de rendimiento comentados anteriormente:

1. **Abuso del objeto de sesión:** Tal vez este sea uno de los principales problemas en las aplicaciones Web. Guardar objetos en la sesión (HttpSession) suele ser el camino más fácil para mantener los

datos a través de las diferentes pantallas que componen una aplicación. Suele ser una práctica habitual entre los programadores más noveles. Almacenar grandes cantidades de información en este objeto provoca un incremento de memoria usada (heap) proporcional al número de usuarios conectados (Recordar que el objeto de sesión esta asociado a cada usuario conectado).

2. **Anidación de consultas SQL**: En muchos equipos de desarrollo, las consultas SQL son escritas por los programadores Java, que en ocasiones no son conscientes de los costos de dichas consultas. En lugar de escribir una SQL que une diferentes tablas, tienden a llamar a las sentencias de SQL en los bucles. Esto aumenta el número de consultas lanzadas para realizar una operación, aumentando considerablemente el tiempo de consulta y provocando una sobrecarga de la base de datos.
3. **Consultas SQL ineficientes:** Se trata de un problema muy común en las aplicaciones. Un proyecto J2EE necesitaría de un experto en SQL capaz de optimizar el rendimiento de las consultas a BD. Una buena práctica es la creación de vistas y procedimientos por un experto en SQL, que aseguran la correcta optimización de las consultas y simplifican la lógica de la aplicación Java.
4. **Código o algoritmos ineficientes:** Es de vital importancia buenas prácticas en la elaboración del código, así como de tener muy en cuenta el rendimiento cuando una aplicación requiera de algoritmos complejos.
5. **Creación de objetos de forma cíclica:** Es habitual encontrarse bucles en los que se crean nuevos objetos en cada ciclo del mismo. Esto es una mala práctica ya que provoca una sobrecarga de la memoria del programa para albergar dichos objetos, que aunque su ciclo de vida no supera una iteración del bucle y son eliminados de memoria, provoca un mayor estrés al "garbage collector" que tendrá que actuar con mayor frecuencia para limpiar la memoria. Recordar que cuando el "garbage collector" entra en acción todos los threads java se detienen hasta que su trabajo termine.

1.1.2. Tipos de análisis de rendimiento

El análisis de rendimiento (profiling) puede clasificarse en dos aspectos: Análisis de CPU y análisis de memoria.

1.1.2.1. Análisis de CPU

En análisis de la CPU se basa en identificar y estudiar la latencia de los métodos de la aplicación. Entendemos por latencia la cantidad de tiempo empleada en ejecutar un método.

La latencia de un método puede ser medida de dos formas: tiempo total transcurrido (elapsed time) y tiempo de CPU:

Elapsed time: Es el tiempo total transcurrido en la ejecución de un método. Esto incluye el tiempo empleado en ejecutar sus "sub-métodos", tiempos de red, tiempos de lectura-escritura en disco. Por ejemplo, consideramos un método de un objeto de acceso a datos (dao), el tiempo de latencia total sería el tiempo empleado en ejecutar el código java, junto con el tiempo empleado en conectarse a BD y el tiempo que tarde la BD en ejecutar y retornar los resultados de la sentencia SQL. Por lo tanto podremos aplicar la siguiente fórmula:

Elapsed time = CPU time + DISK I/O + Network I/O

CPU time: El tiempo de CPU se refiere al tiempo de CPU empleado por un método, función o rutina para ejecutar su lógica. Es decir, no tenemos en cuenta las demoras provocadas por las entradas y salidas del sistema (disco o red). No tenemos en cuenta las interrupciones al procesador producidas por dichas entradas y salidas.

CPU time = Time exclusively spent on CPU (without I/O or any Interrupts delay)

Con la comparación de estas dos métricas para la ejecución de un método podemos obtener algunas conclusiones:

- Si el tiempo de CPU es muy elevado podemos deducir que el método requiere de un procesamiento intenso y no requiere de I/O de disco o de red. Puede que la lógica de nuestro programa sea demasiado compleja, o que estemos dejando muchos objetos en memoria que requieren del Garbage Collector.
- Si el tiempo total de ejecución del método es claramente superior al tiempo de CPU, significa que estamos haciendo fuerte uso de utilidades de red o de disco.

1.1.2.2. Análisis de memoria

El análisis de memoria se refiere al estudio de cómo se crean los objetos en la memoria de la JVM en tiempo de ejecución. Mediante este análisis podemos identificar diversos problemas en la aplicación, especialmente provocados por las fugas de memoria (memory leak).

1.2 Introducción a la herramienta HP Diagnostics

HP Diagnostics supervisa la salud de las aplicaciones tradicionales, que permite el aislamiento rápido y resolución de problemas. Se trata de una herramienta que permite su uso en el ciclo completo de la aplicación.

Como cualquier herramienta de profiling de aplicaciones Java dispone de dos componentes:

- Agent: El agente también denominado “Probe” o “Profiler” el cual se ejecutará en el servidor que se desea analizar. El agente ataca a las métricas de la máquina virtual usando las APIs JVMPI/JVMTI, enviando dichos indicadores a otro componente (consola o servidor de profiling) fuera del servidor que se encargará de recolectarlos.
- Console: Es el componente encargado de recolectar los datos enviados por el agente y representarlos de forma que puedan ser fácilmente estudiadas. En el caso de la herramienta HP Profiling, la consola puede ser un servidor central al que todos los agentes le envían las métricas o una especie de servidor local que se instalará en la misma máquina que el agente. Este documento se centrará en la consola de tipo local.

En la siguiente imagen se representa la arquitectura general de una herramienta de profiling:

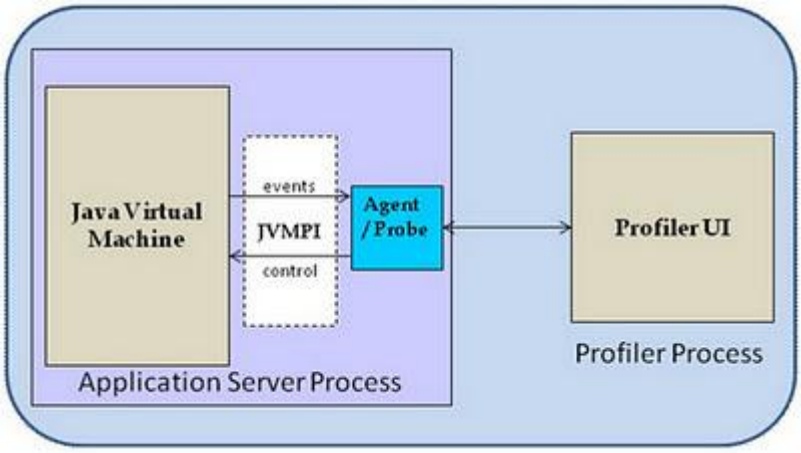


Figura 1.3 – Arquitectura de la herramienta de análisis

La herramienta de profiling puede ser usada sobre servidores de aplicaciones o sobre aplicaciones ejecutadas en “standalone”, para ello el agente tiene que estar ligado a la máquina virtual. Este proceso es denominado “Agent Integration” o “Profiler Integration”. Este proceso es llevado a cabo a través de ciertos parámetros en el arranque de la máquina virtual y en el caso de los servidores de aplicaciones, también en sus script de inicio. En la siguiente figura podemos ver el diagrama de actividades en una herramienta de profiling:

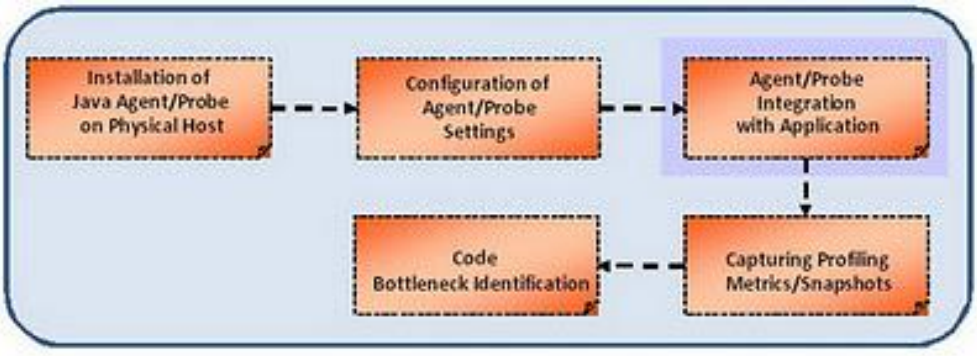


Figura 1.4– Diagrama de actividades de la herramienta de análisis

Destacar de la imagen anterior el paso de configuración del Agente, donde es posible personalizar las capas, clases, método o componentes que se van a instrumentalizar (componentes en los que el agente interceptará su bytecode para obtener las métricas llevadas a estudio). Es muy importante tener en cuenta que dicho proceso de instrumentalización es intrusivo y por lo tanto provoca una mayor carga de la máquina tanto a nivel de JVM como a nivel de hardware. Es decir, los dos datos mostrados por la herramienta no son al 100% reales ya que si desactivamos el agente probablemente nuestra aplicación tendrá un mayor rendimiento. Por lo tanto, contra más componentes deseemos instrumentar peor será su rendimiento, siendo esta relación generalmente proporcional. Este documento parte de la configuración detallada en el manual XXXX, siendo dicha configuración mínimamente intrusiva afectando en menor medida al rendimiento de la aplicación y proporcionando un conjunto de información suficiente para detectar anomalías en la misma.

2 Usando HP Diagnostics

Si han seguido los pasos de la guía de configuración XXXX, una vez levantado el servidor de aplicaciones Weblogic 11, podremos acceder a la consola para el estudio de las aplicaciones desplegadas en el a través de la siguiente URL: http://localhost:35000/

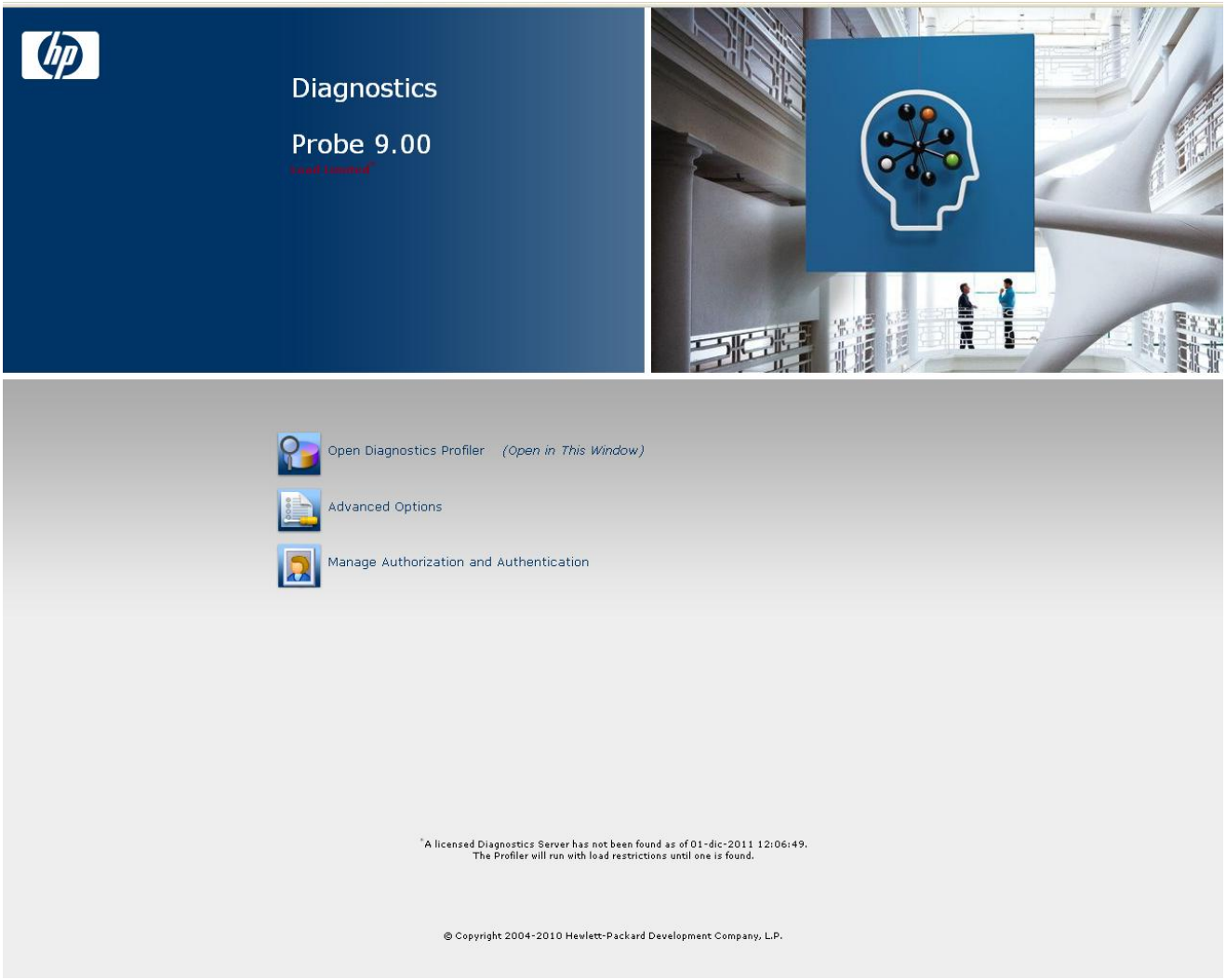


Figura 2.1– Pantalla inicial

Haciendo clic sobre “Open Diagnostics Profiler” se cargará el applet principal de análisis de rendimiento que estará compuesto por diferentes pestañas, las cuales recorreremos a lo largo de este apartado.

Nota: Dependiendo si estamos intrumentalizando una máquina virtual 1.4 ó una JDK 6 (weblogic 8 ó weblogic 11), existen diferencias notables en la herramienta de análisis. Algunas de las opciones disponibles para la JDK 6 no existen para la JDK 1.4. Esto es debido a que las APIs de instrumentación de la JDK 1.4 eran bastante limitadas y han ido evolucionando a lo largo de las diferentes versiones de la máquina virtual. En este documento se ilustrarán a través de notas como esta, las opciones que difieren o no existen en caso de estar realizando un análisis de una aplicación sobre weblogic 8.

2.1 Pestaña Summary – Visión general

Al cargar el applet, la primera pestaña que se presenta es la de resumen que nos proporciona una visión general del estado de las aplicaciones desplegadas en el servidor de aplicaciones:

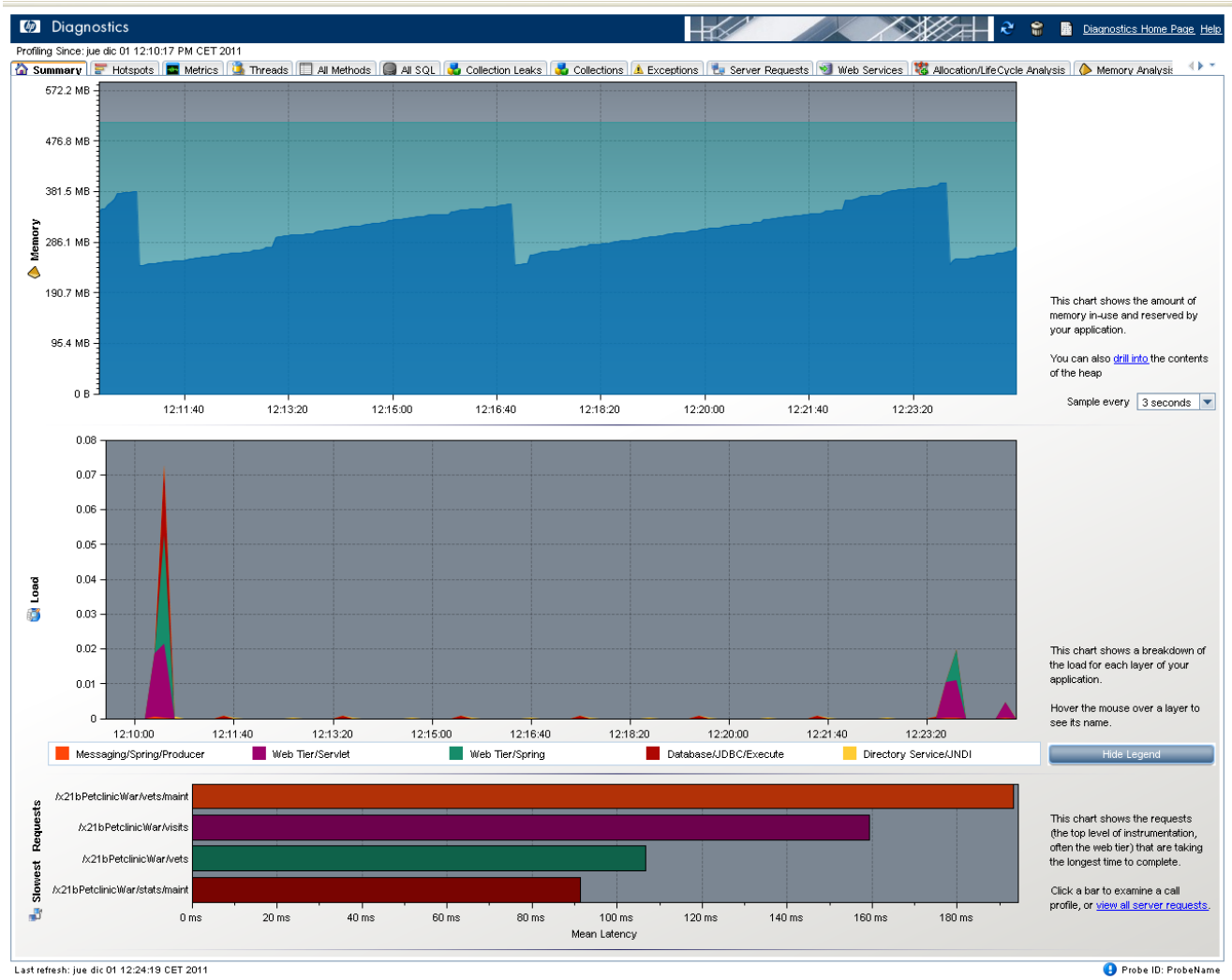


Figura 2.2– Pestaña de resumen

Como se puede observar en la imagen superior, esta pantalla se divide en tres secciones:

- **Memoria:** Proporciona un grafico de la cantidad de memoria consumida en un intervalo de tiempo.
- **Carga:** Proporciona un gráfico de la carga del servidor. Por ejemplo si se realiza una petición http se mostrará en este gráfico en forma de pico. Dichos picos estará divididos en capas representadas en diferentes colores. Por ejemplo, para una petición http puede aparecer la capa del servlet que atiende la petición, la capa de spring y la capa de acceso a datos.
- **Peticiones más lentas:** En el gráfico inferior podemos encontrar las peticiones http más lentas (representadas por su patrón URL de entrada). En apartado se representa la latencia de las peticiones consideradas más lentas.

2.2 Pestaña Hotspots

Esta pantalla también nos proporciona un resumen de los procesos considerados más costosos.

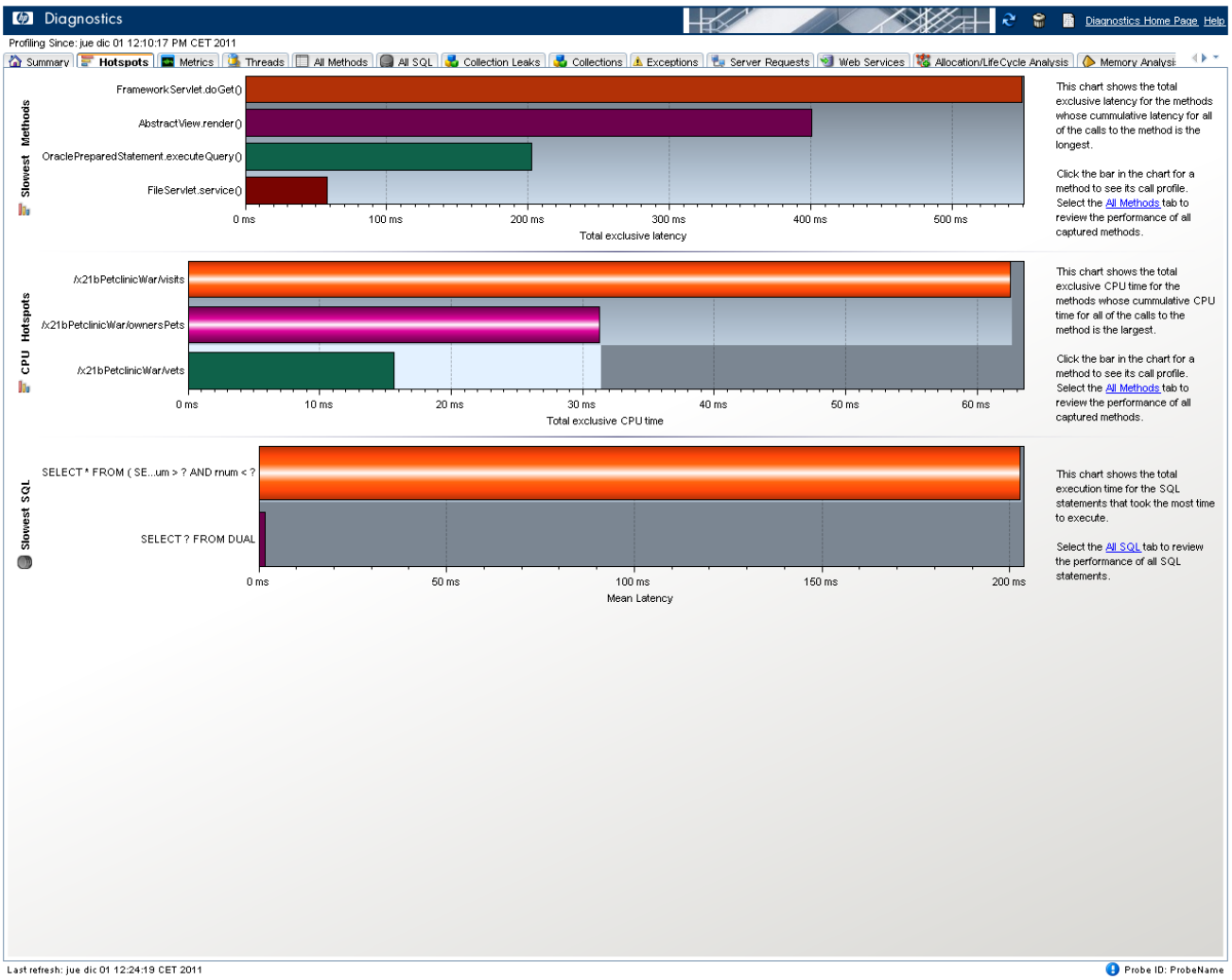


Figura 2.3– Pestaña Hotspots

Se divide en tres secciones:

- **Métodos más lentos:** Nos proporciona una visión de los métodos con mayor latencia.
- **Tiempos de CPU:** Nos proporciona una visión de los tiempos de CPU para las peticiones http más costosas. Dichas peticiones son representadas por su patrón de URL de entrada al servidor.
- **SQL más lentas:** Nos proporciona un resumen gráfico de las SQL más costosas.

2.3 Pestaña Metrics

Desde esta pantalla podemos ver algunos datos proporcionados por la intrumentalización del sistema operativo, la máquina virtual y Weblogic. En el caso de Weblogic dichas métricas son extraídas a través de los MBeans.

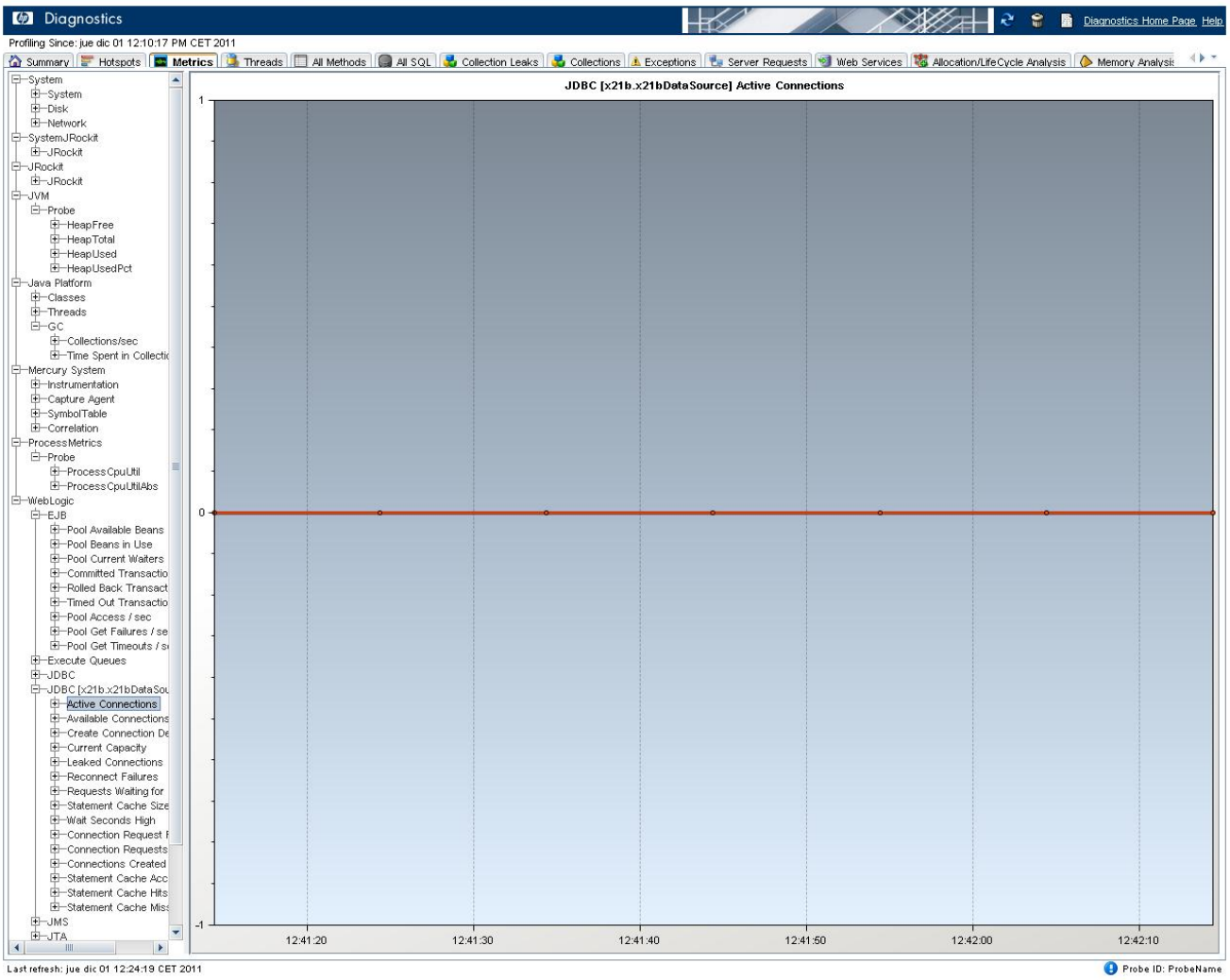


Figura 2.4– Pestaña Metrics

Entre la variedad de métricas disponibles en la estructura tipo árbol, destacamos las siguientes:

- **System**
  - **System:** Podemos encontrar métricas interesantes como la CPU y la memoria del equipo.
  - **Disk:** Rendimiento de los discos del equipo. Velocidad de lectura/escritura.
  - **Network:** Rendimiento de entrada y salida de datos a nivel de red.
- **JRockit**: Métricas sobre el rendimiento de la máquina virtual de JRockit
- **JVM:** Métricas relacionadas con el HEAP.
- **Java Platform**
  - **Classes:** Se pueden analizar el número de clases cargadas y como se destruyen.
  - **Thread:** Métricas relacionadas con el número de threads.

- **GC:** Funcionamiento del Garbage Collector.

- **Weblogic (11)**
  - **EJB:** Diferentes EJBs cargados, fallos de los mismos, etc…
  - **JDBC:** Podemos ver las características de un pool de conexiones en concreto.
  - **JMS:** Colas JMS.
  - **JTA:** Datos referentes a Java Transaction API.
  - **Web Sessions:** Número de sesiones http activas.
  - **Servlets:** Servlets cargados en el sistema.
  - **Thread Pool:** Pool de threads de Weblogic. Estos threads son los que se encargarán de atender las diferentes peticiones http.
  - **Web Services:** Métricas relacionadas con los servicios web expuestos e invocados.
- **Weblogic (8)**
  - **EJB:** Diferentes EJBs cargados, fallos de los mismos, etc..
  - **Execute Queues:** Threads del Weblogic. Peticiones atendidas. Numero de peticiones atendidas en un intervalo de tiempo.
  - **JMS:** Colas JMS.
  - **JTA:** Datos referentes a Java Transaction API.
  - **Web Sessions:** Número de sesiones http activas.
  - **Servlets:** Servlets cargados en el sistema.

Nota: En esta pestaña encontramos las métricas del servidor de aplicaciones que deberán situarse entre los valores especificados por los indicadores establecidos por el departamento de Calidad para la certificación (CERT-ISA). Estos indicadores se pueden consultar en el apartado “3 Pruebas de prestaciones” localizados en el documento “Procedimiento de evaluación de indicadores en Desarrollo”.

Nota: Las métricas mostradas para Weblogic 11 difieren de las métricas asociadas a Weblogic 8.

2.4 Pestaña Threads

Desde esta pantalla podremos visualizar los diferentes threads abiertos por el weblogic, tanto los abiertos para atender las peticiones, como los threads internos para su propia gestión. Una característica interesante de esta pestaña es la posibilidad de realizar un volcado de threads, para así poder estudiar cuales están a la espera, que operaciones realizan o tal vez cuales se han quedado bloqueados y por qué.

Nota: Esta opción no está disponible para Weblogic 8.

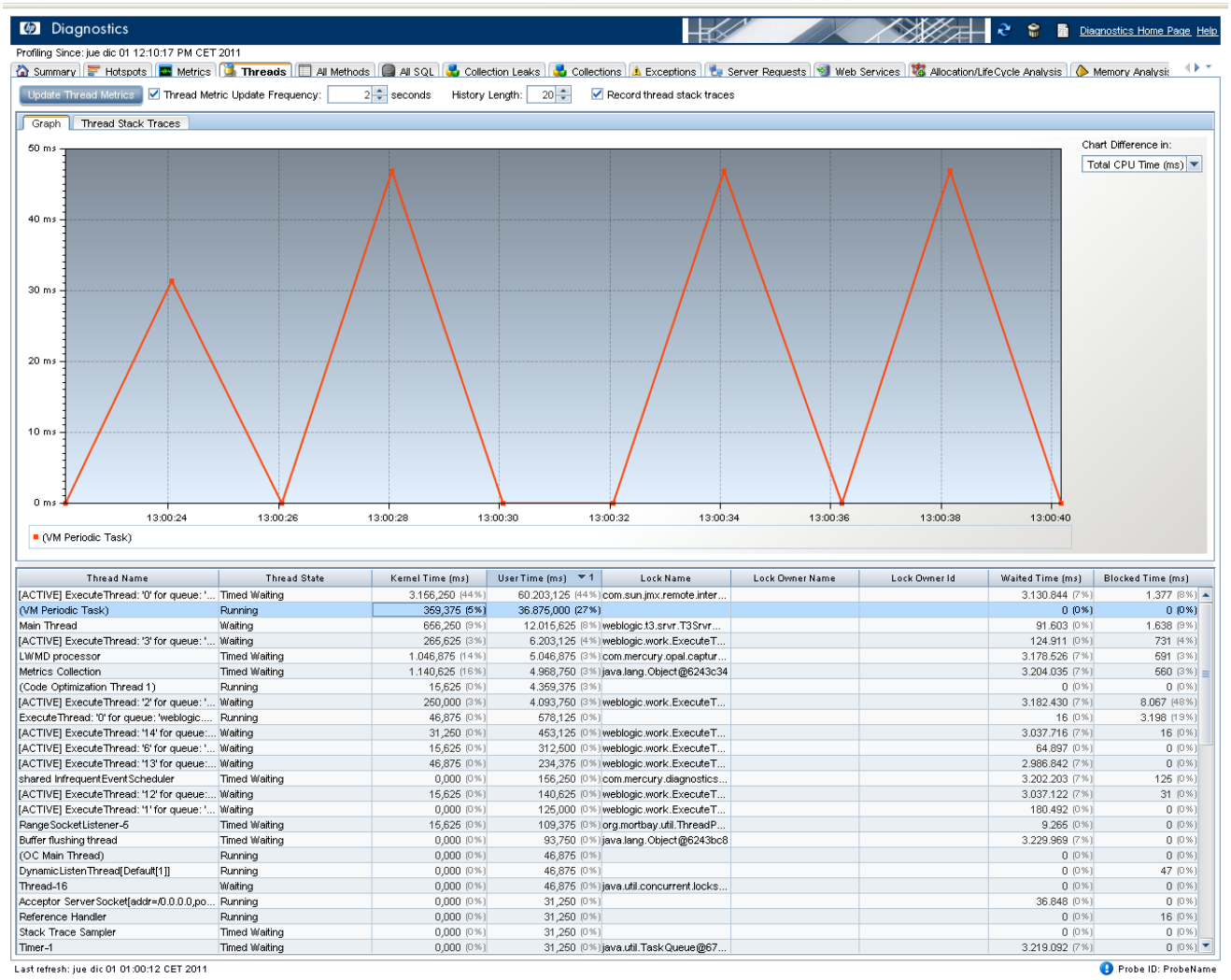


Figura 2.5– Pestaña Threads

Si accedemos a la subpestaña de “Thread Stack Traces” podemos realizar un volcado de threads:

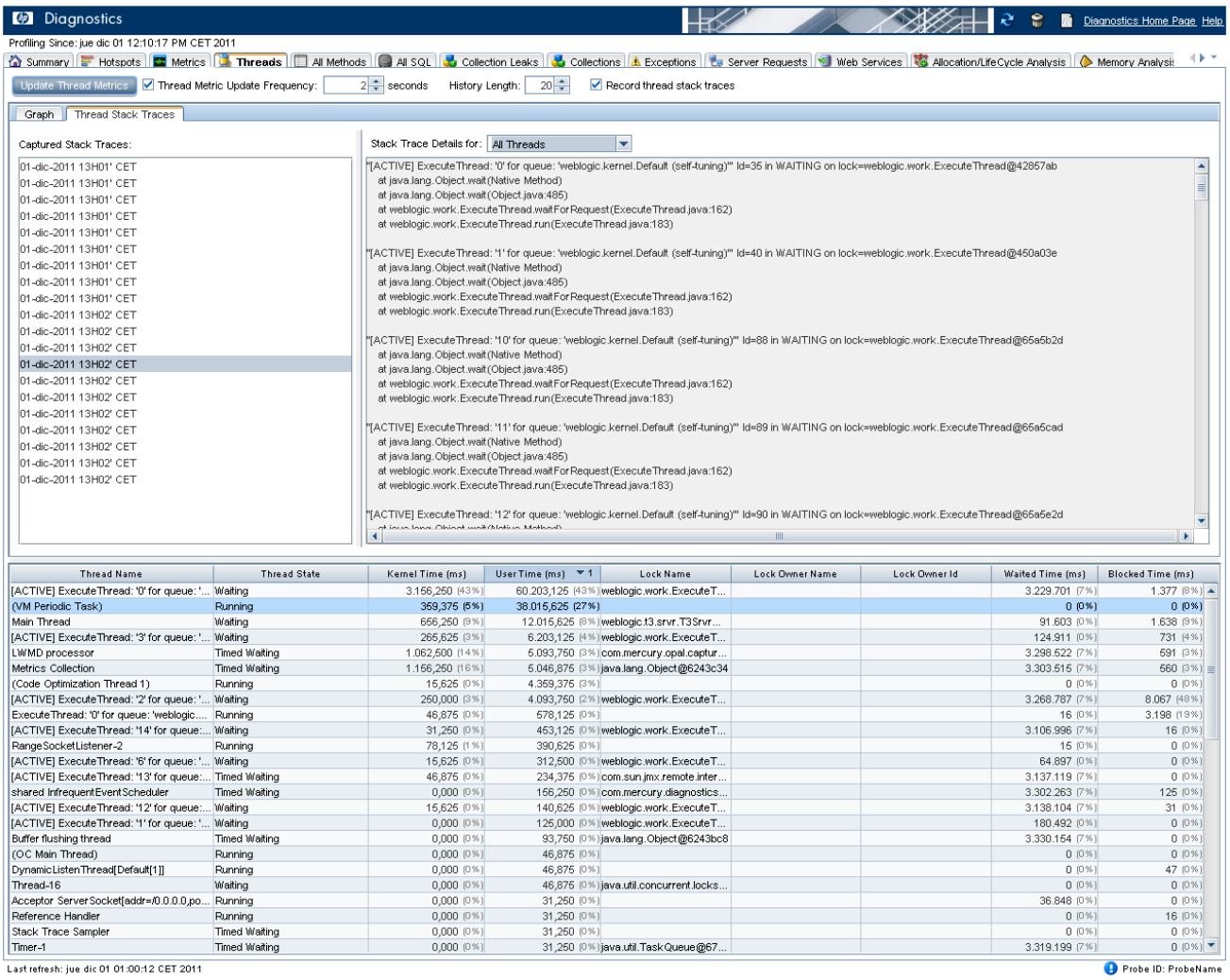


Figura 2.6– Volcado de Threads

2.5 Pestaña All Methods

En las dos pestañas iniciales podíamos ver los métodos considerados como más costosos o de mayor latencia, en esta pantalla podremos ahondar más en todos ellos.

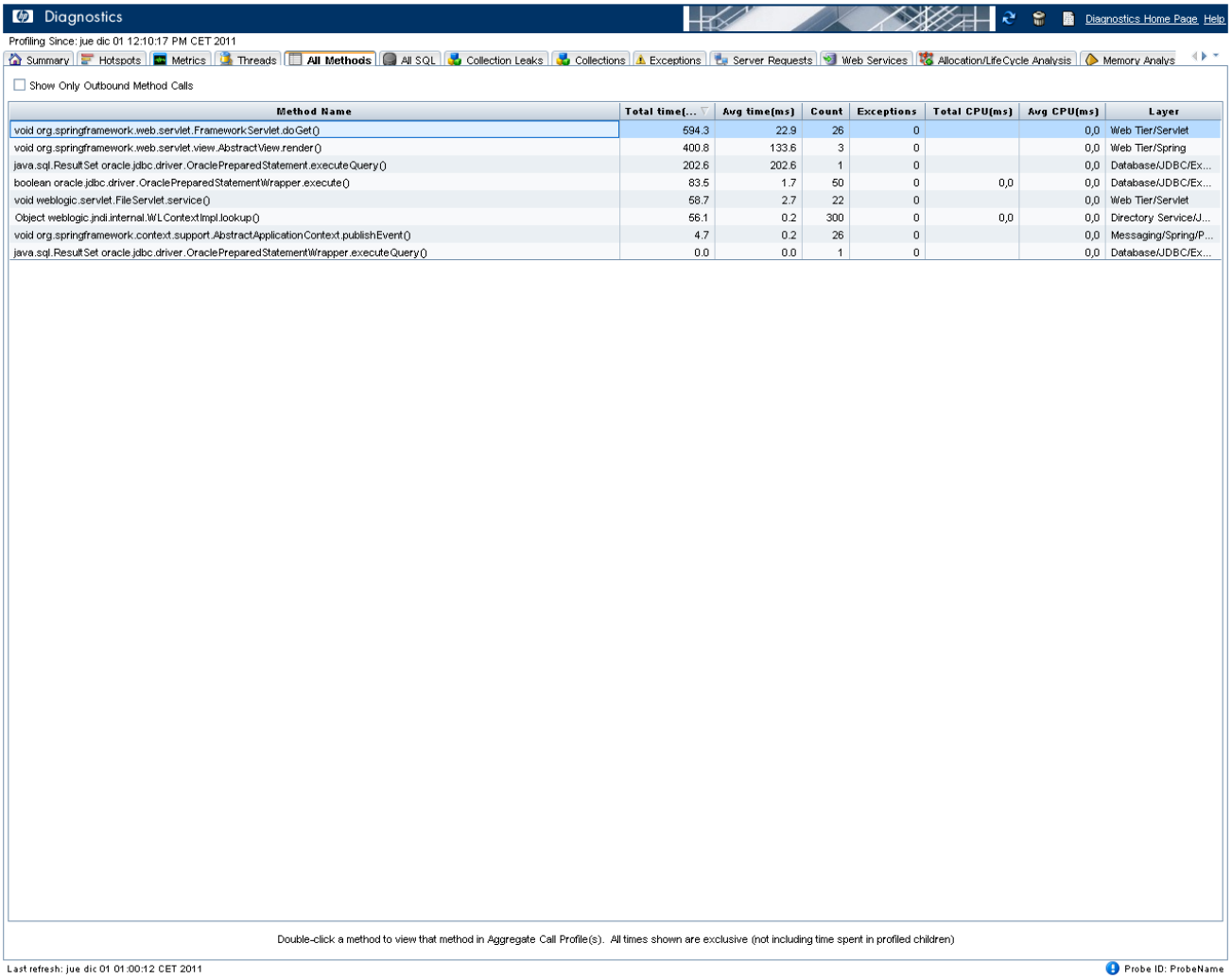
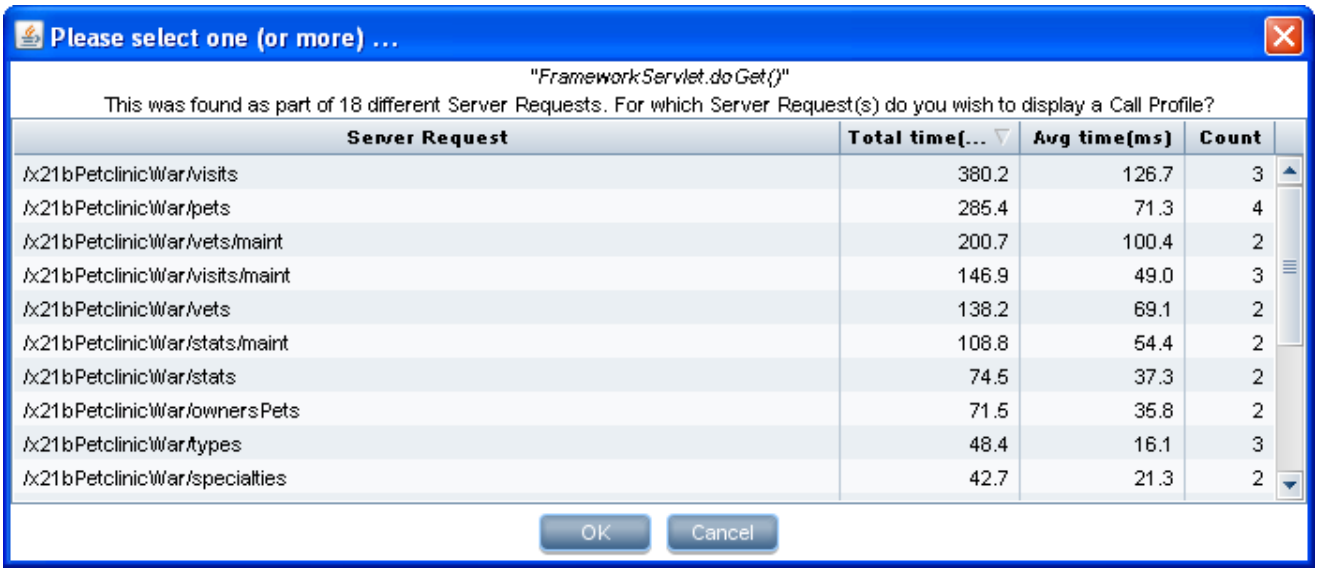


Figura 2.7– Pestaña All methods

En la tabla resumen se muestran los siguientes datos:

- **Tiempo total:** Tiempo total consumido en la ejecución del método que ha sido invocado N veces.
- **Tiempo medio:** Tiempo total / numero de invocaciones. Tiempo medio de latencia.
- **Numero de invocaciones:** Numero de veces que se ha invocado dicho método.
- **Excepciones:** Numero de excepciones que ha producido la ejecución del método.
- **Tiempo total de CPU:** Tiempo total de CPU consumido en las N ejecuciones del método.
- **Capa:** Capa a la que pertenece el método. Las capas son las mismas que se mostraron en la pestaña “summary”.

Frecuentemente en esta tabla resumen aparecerán métodos relacionados con la envoltura de servlets de Spring. La herramienta esta preparada para poder analizar dichos envoltorios de spring, así que si hacemos doble clic en una estas filas, aparecerá un resumen de las peticiones a las que ha atendido dicho servlet:



Server Request	Total time[...	Avg time[ms]	Count
/x21bPetclinicWar/visits	380.2	126.7	3
/x21bPetclinicWar/pets	285.4	71.3	4
/x21bPetclinicWar/vets/maint	200.7	100.4	2
/x21bPetclinicWar/visits/maint	146.9	49.0	3
/x21bPetclinicWar/vets	138.2	69.1	2
/x21bPetclinicWar/stats/maint	108.8	54.4	2
/x21bPetclinicWar/stats	74.5	37.3	2
/x21bPetclinicWar/ownersPets	71.5	35.8	2
/x21bPetclinicWar/types	48.4	16.1	3
/x21bPetclinicWar/specialties	42.7	21.3	2

Figura 2.8– Peticiones atendidas por un servlet

En cambio si pulsamos sobre alguno de los métodos no considerados como servlets, podemos visualizar su pila de llamadas de forma gráfica:

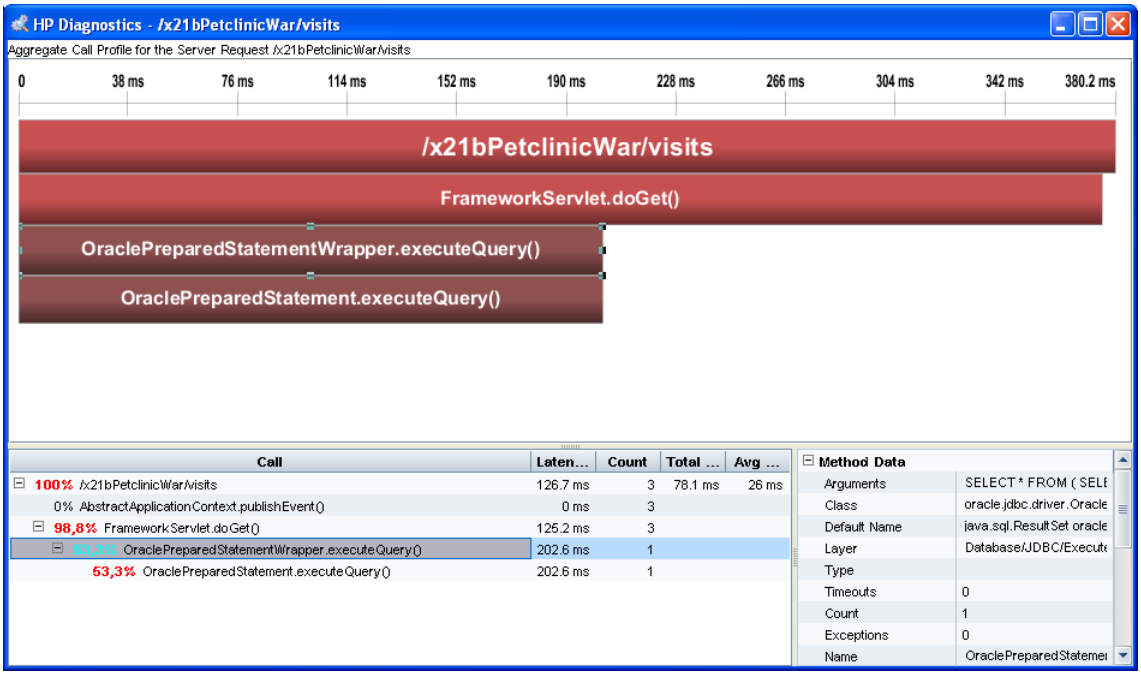


Figura 2.9– Pila de invocación de un método

2.6 Pestaña All SQL

Desde esta pantalla podemos ver las estadísticas de las SQL lanzadas por la aplicación. No debe llevarnos a equívoco el nombre de esta pestaña, ya que en ella solo se representarán las sentencias SQL más costosas:

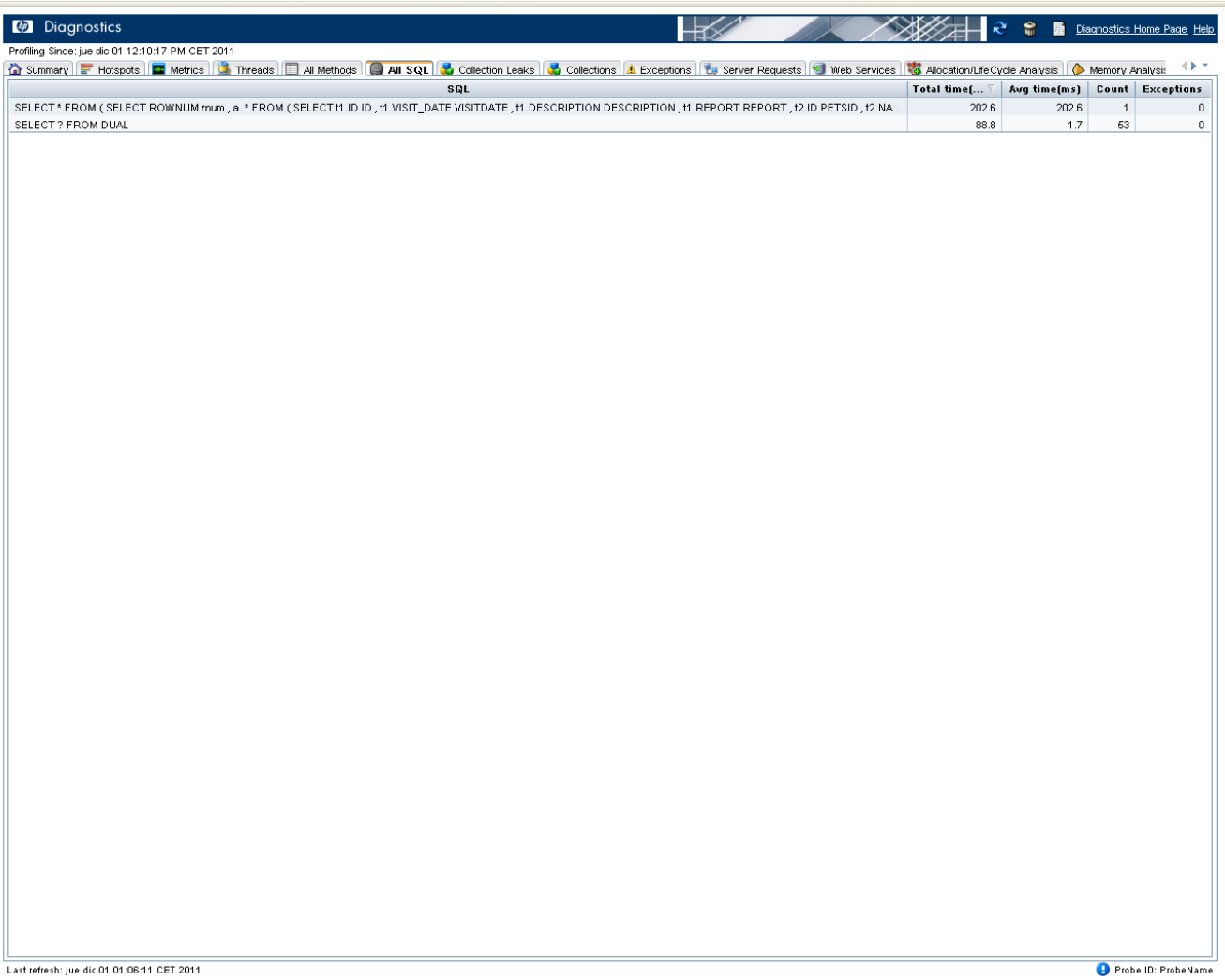
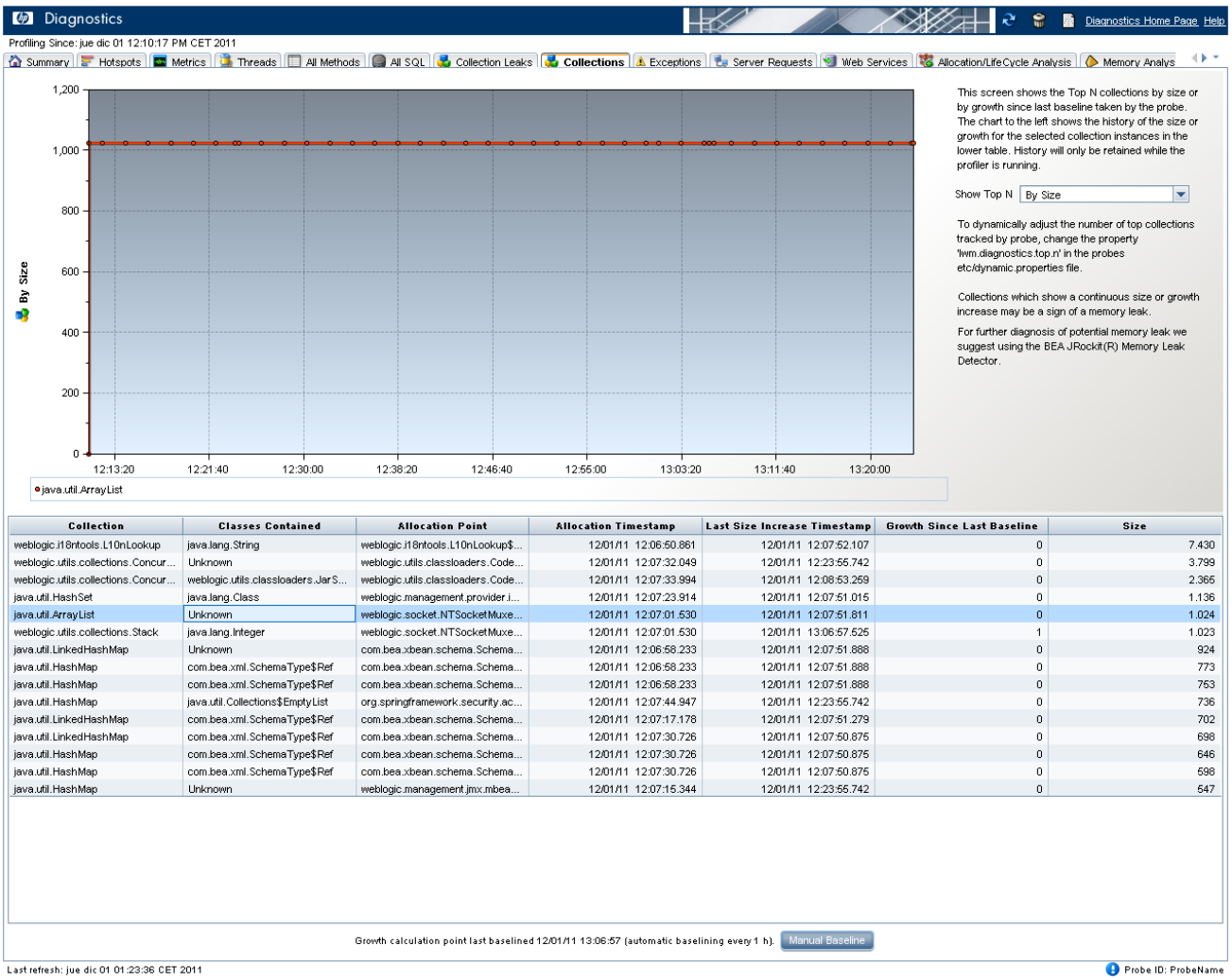


Figura 2.10– Pestaña All SQL

2.7 Pestaña Collection

Un típico de problema de rendimiento de las aplicaciones Java suele ser el abuso de las colecciones de objetos o el no elegir correctamente el tipo de colección adecuada (depende si queremos ordenarla, si deseamos asociarle una clave, sincronizadas o no sincronizadas, etc..). Desde esta pantalla podemos visualizar las colecciones que residen en la JVM que contienen, cuantos elementos, desde donde han sido creadas y cuanta memoria ocupan:



Collection	Classes Contained	Allocation Point	Allocation Timestamp	Last Size Increase Timestamp	Growth Since Last Baseline	Size
weblogic.i18ntools.L10nLookup	java.lang.String	weblogic.i18ntools.L10nLookup$...	12/01/11 12:06:50.861	12/01/11 12:07:52.107	0	7.430
weblogic.utils.collections.Concur...	Unknown	weblogic.utils.classloaders.Code...	12/01/11 12:07:32.049	12/01/11 12:23:55.742	0	3.799
weblogic.utils.collections.Concur...	weblogic.utils.classloaders.JarS...	weblogic.utils.classloaders.Code...	12/01/11 12:07:33.994	12/01/11 12:08:53.259	0	2.365
java.util.HashSet	java.lang.Class	weblogic.management.provider.i...	12/01/11 12:07:23.914	12/01/11 12:07:51.015	0	1.136
java.util.ArrayList	Unknown	weblogic.socket.NTSocketMuxe...	12/01/11 12:07:01.530	12/01/11 12:07:51.811	0	1.024
weblogic.utils.collections.Stack	java.lang.Integer	weblogic.socket.NTSocketMuxe...	12/01/11 12:07:01.530	12/01/11 13:06:57.525	1	1.023
java.util.LinkedHashMap	Unknown	com.bea.xbean.schema.Schema...	12/01/11 12:06:58.233	12/01/11 12:07:51.888	0	924
java.util.HashMap	com.bea.xml.SchemaType$Ref	com.bea.xbean.schema.Schema...	12/01/11 12:06:58.233	12/01/11 12:07:51.888	0	773
java.util.HashMap	com.bea.xml.SchemaType$Ref	com.bea.xbean.schema.Schema...	12/01/11 12:06:58.233	12/01/11 12:07:51.888	0	753
java.util.HashMap	java.util.Collections$EmptyList	org.springframework.security.ac...	12/01/11 12:07:44.947	12/01/11 12:23:55.742	0	736
java.util.LinkedHashMap	com.bea.xml.SchemaType$Ref	com.bea.xbean.schema.Schema...	12/01/11 12:07:17.178	12/01/11 12:07:51.279	0	702
java.util.LinkedHashMap	com.bea.xml.SchemaType$Ref	com.bea.xbean.schema.Schema...	12/01/11 12:07:30.726	12/01/11 12:07:50.875	0	698
java.util.HashMap	com.bea.xml.SchemaType$Ref	com.bea.xbean.schema.Schema...	12/01/11 12:07:30.726	12/01/11 12:07:50.875	0	646
java.util.HashMap	com.bea.xml.SchemaType$Ref	com.bea.xbean.schema.Schema...	12/01/11 12:07:30.726	12/01/11 12:07:50.875	0	598
java.util.HashMap	Unknown	weblogic.management.jmx.mbea...	12/01/11 12:07:15.344	12/01/11 12:23:55.742	0	547

Figura 2.11– Pestaña Collections

2.8 Pestaña Exceptions

Esta pestaña proporciona un punto interesante desde el cual analizar las excepciones no controladas producidas en el programa:

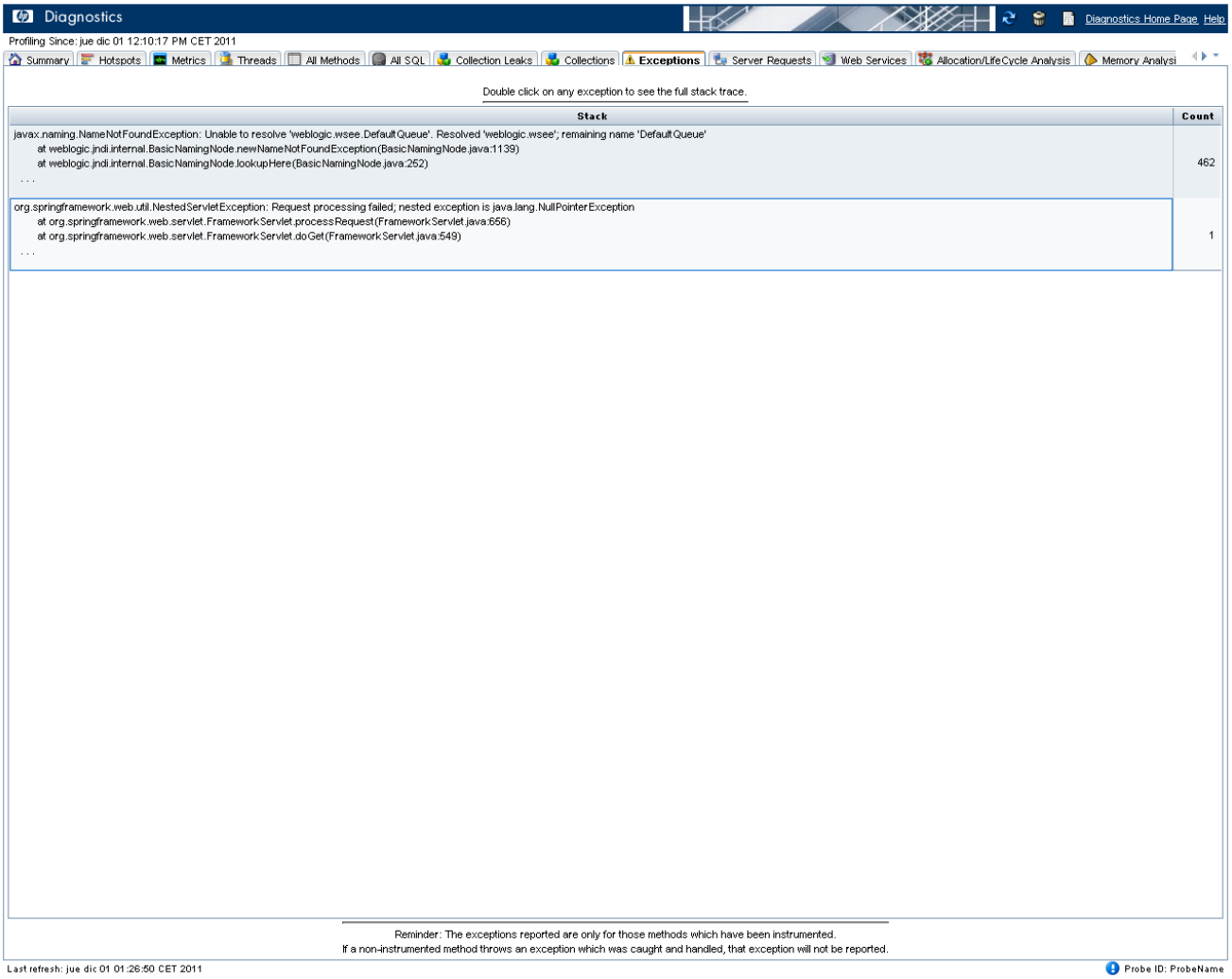


Figura 2.12– Pestaña Exceptions

Haciendo doble clic sobre una de las filas de las excepciones podemos ver su pila:

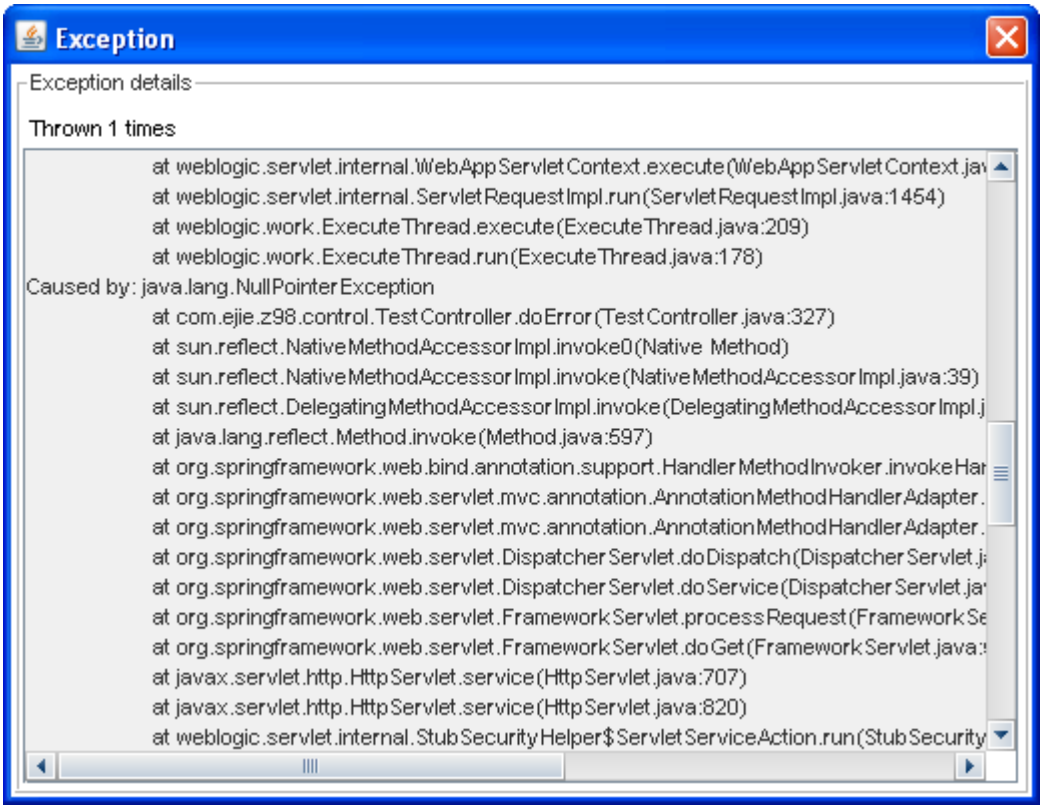


Figura 2.13– StackTrace de la excepción

2.9 Pestaña Server Request

Desde esta pantalla se pueden visualizar todas las peticiones http realizadas al servidor (identificadas por su patrón URL). Podemos analizar el número de peticiones, sus latencias y tiempo medio de CPU. Al igual que otras pestañas podemos profundizar en la pila de llamadas asociadas a una petición:

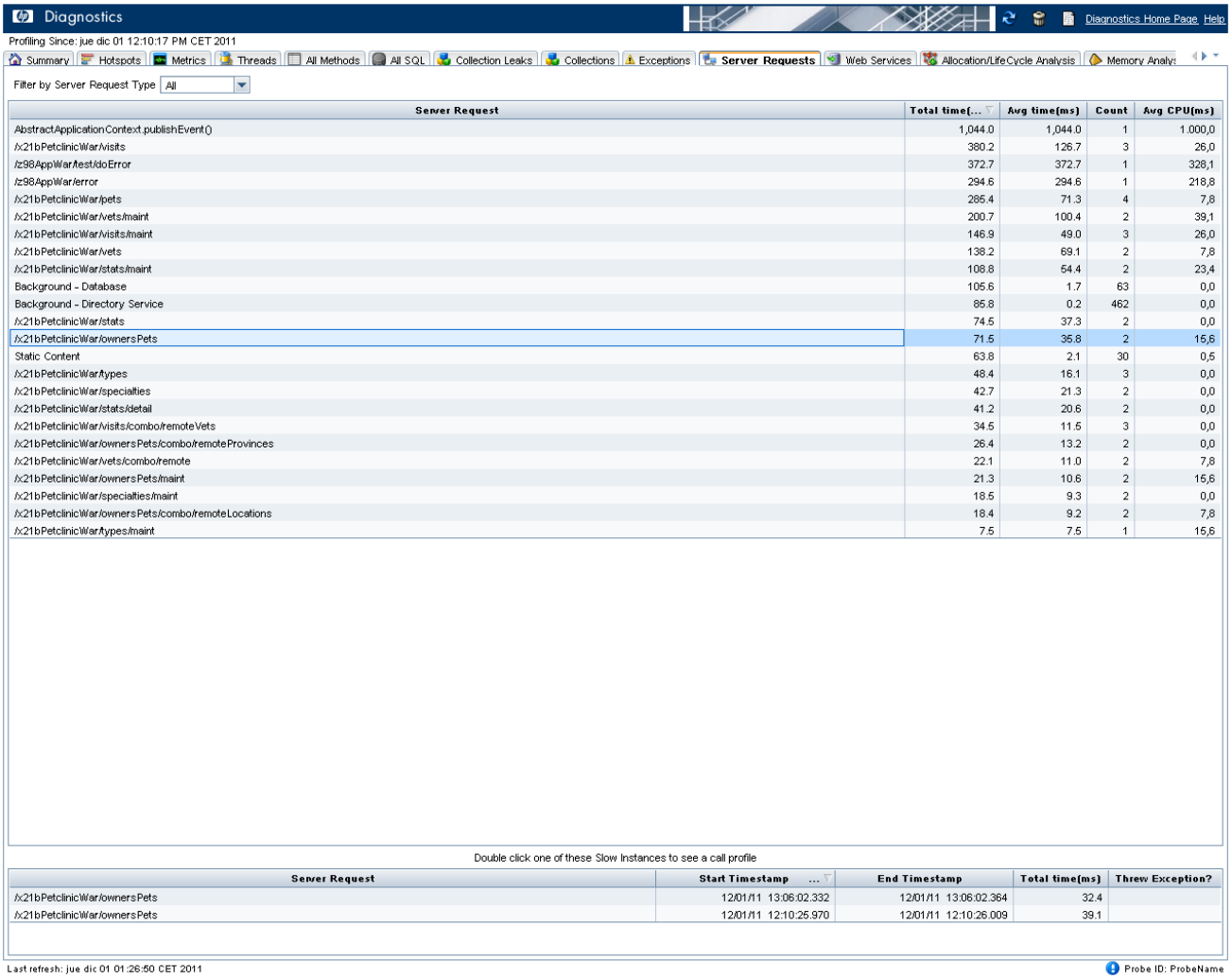


Figura 2.14– Pestaña Server Requests

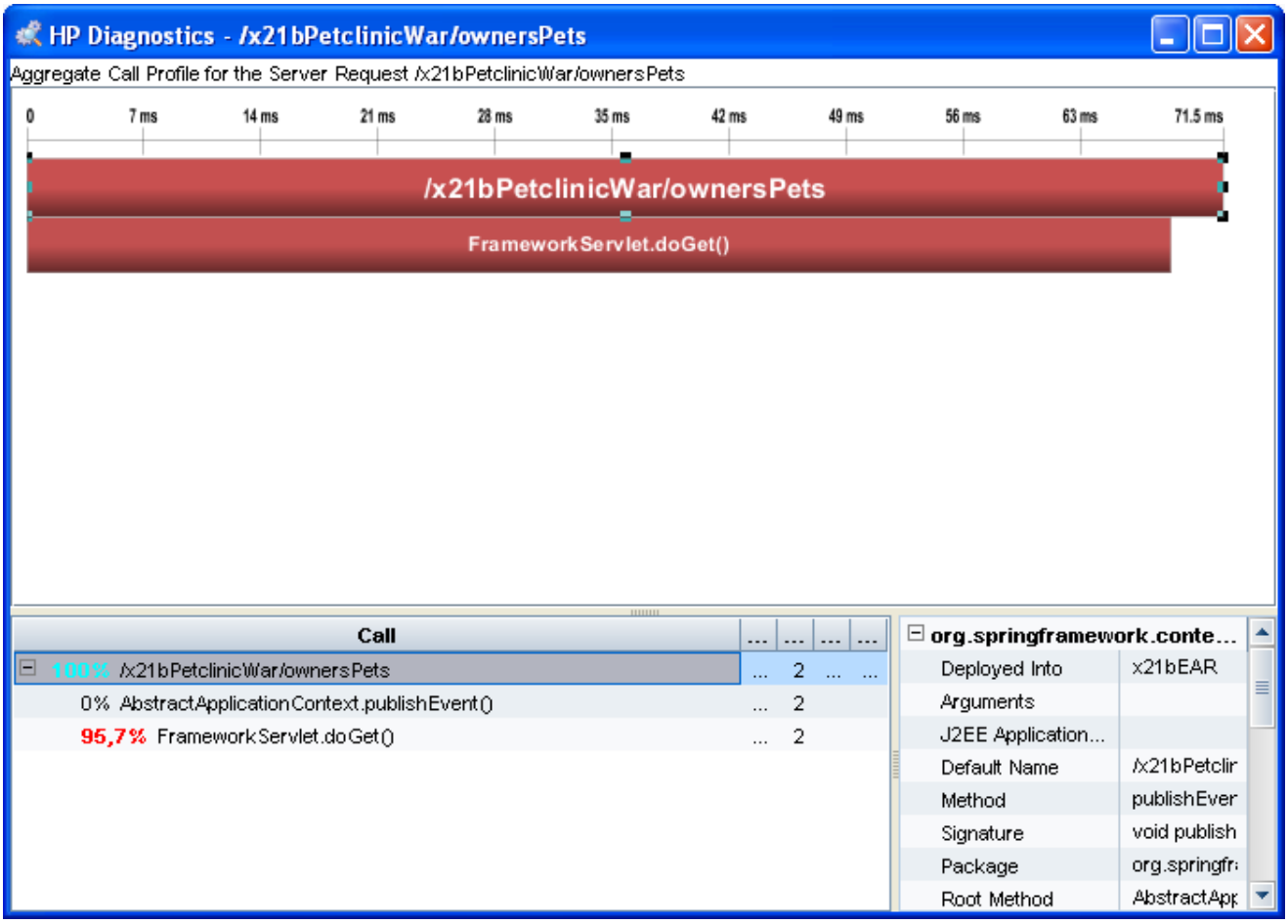


Figura 2.15– Pila de una petición

2.10 Pestaña Web Services

Desde esta pantalla podemos analizar los tiempos de respuesta de los servicios Web expuestos en una aplicación. También será posible acceder a las estadísticas de las invocaciones a otros servicios web remotos, tal y como se muestra en la siguiente figura:

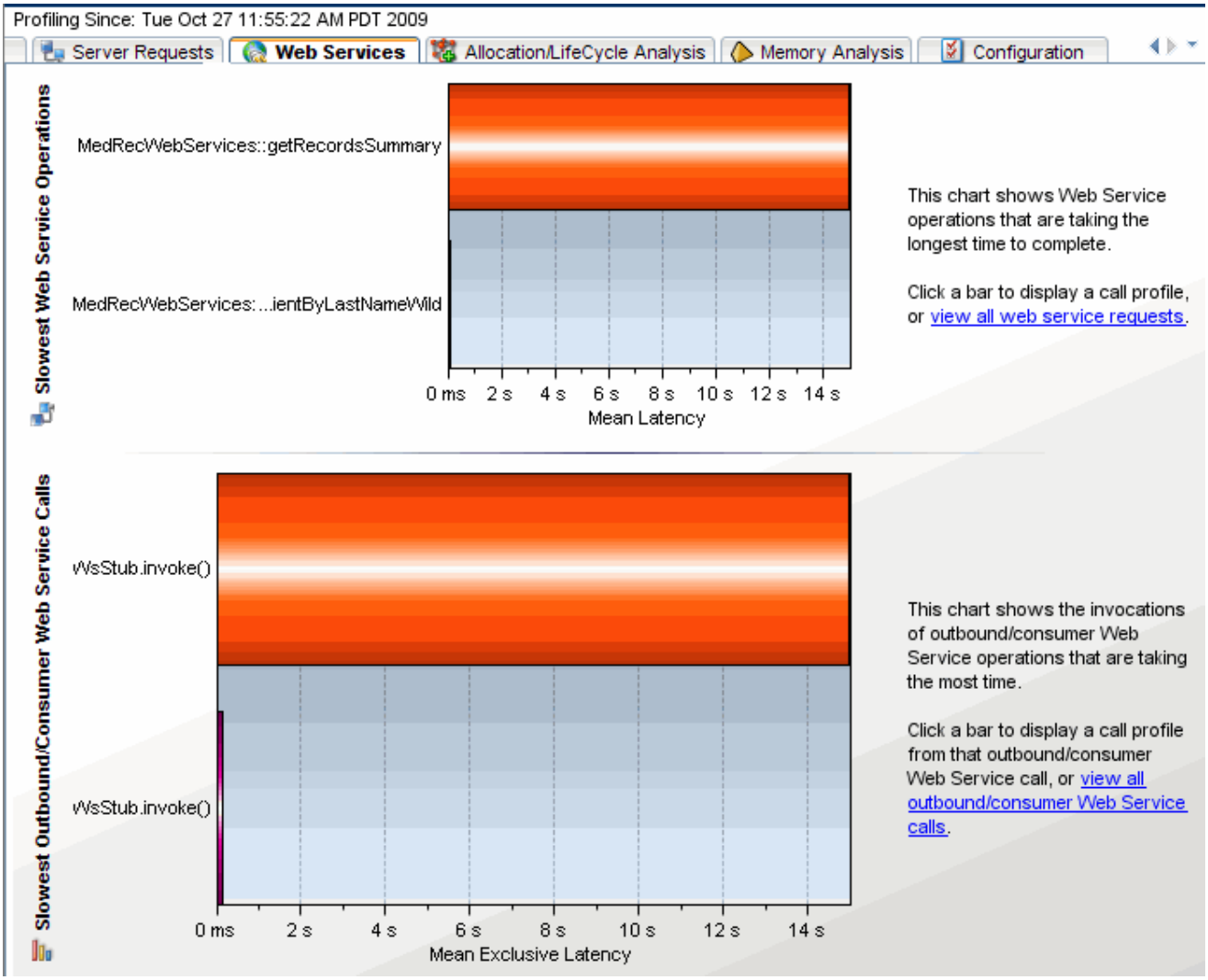


Figura 2.16– Pila de una petición

2.11 Pestaña Memory Análysis

Desde esta pantalla podemos “sacar fotos” del estado actual de los objetos en memoria de la JVM. Para ello pulsaremos sobre el icono  que finalmente mostrará una ventana similar a la siguiente:

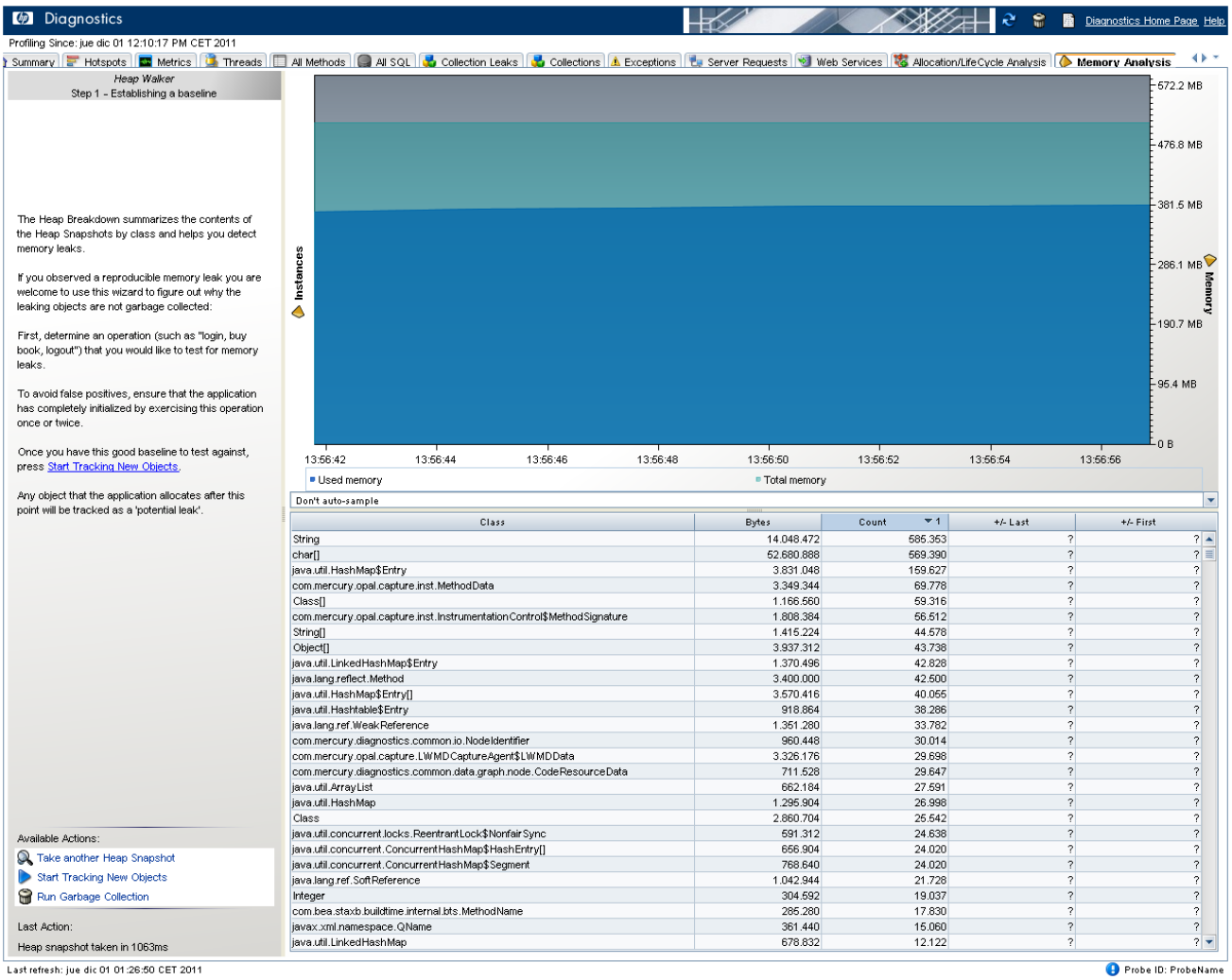


Figura 2.17– Pestaña Memory Analysis

En la parte superior de la pantalla podemos ver un gráfico que muestra la cantidad de memoria consumida por la JVM. En la parte inferior podemos ver una tabla donde se refleja todos los objetos creados en memoria en el momento que ser realizó “la foto”, así como el número de instancias del mismo y el espacio utilizado para almacenarlas.

Esta pestaña tiene la peculiaridad de poder sacar diferentes fotos de la memoria para posteriormente compararlas entre sí y localizar el aumento del número de instancias de los objetos.

Nota: Para la JDK 1.4 (Weblogic 8) esta pestaña se denomina Heap Breakdown y su funcionamiento es minimamente diferente al aquí detallado. Hay que tener en cuenta que el API usado para mostrar los datos en esta pestaña está en un estado experimental en la maquina virtual 1.4 y puede provocar problemas en el rendimiento del servidor así como su bloqueo. Esta opción por defecto se encuentra desactivada y requiere de una configuración especial definida en el documento de instalación de la herramienta.

3 Casos prácticos de uso

En este apartado se presentan una serie de casos que no siguen las buenas prácticas de programación J2EE y que comúnmente se dan en las aplicaciones. Mediante dichos ejemplos aprenderemos a detectar los errores más comunes mediante HP Diagnostics, analizando en que medida afectan al rendimiento de la aplicación, para así poder encontrar una solución a los mismos.

3.1 Buscando objetos en la sesión HTTP

Tal y cómo se mencionaba en los apartados iniciales del presente documento, en numerosas ocasiones se realiza un uso abusivo del objeto de sesión, utilizándolo de forma indiscriminada a modo de almacén de datos que viajan a través de las diferentes pantallas de la aplicación.

El caso práctico que se presenta consiste en un método del controlador Spring que almacena objetos de gran tamaño en la sesión del usuario. Tal vez HP Diagnostics no nos indicará directamente el tamaño del objeto HttpSession, pero si que nos permite deducirlo a través de una serie de pruebas.

El primer síntoma de un uso indebido de sesión es el aumento de la memoria consumida por cada nuevo usuario que accede a la aplicación. Partiremos inicialmente de un estado óptimo de nuestro servidor, al cual todavía no ha accedido ningún usuario. Para asegurarnos de que la memoria de la máquina virtual esta completamente limpia, podemos pulsar sobre el icono  para que se invoque el recolector de basura. Después de realizar estas acciones, obtendremos un gráfico de memoria similar al siguiente:

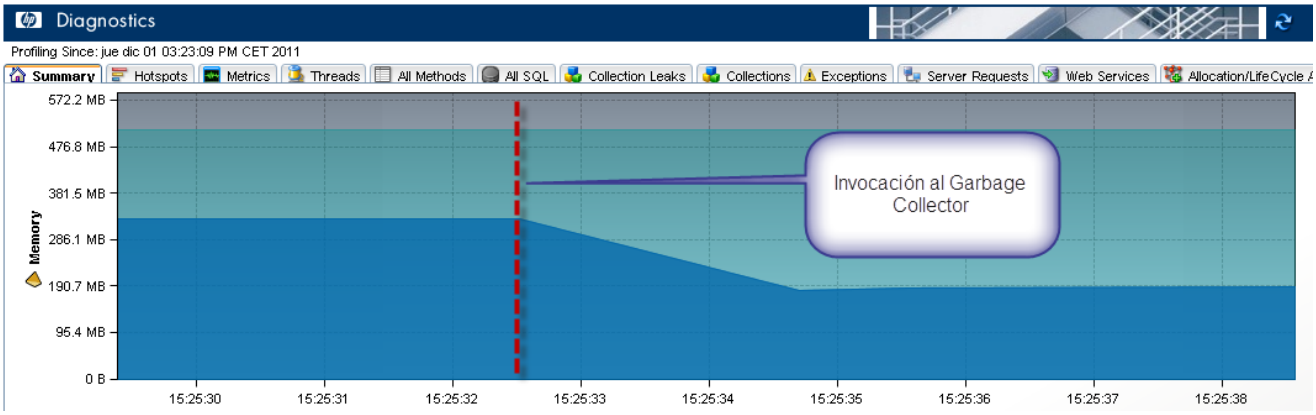


Figura 3.1– Uso de memoria de la pestaña Summary

También nos puede resultar de gran ayuda examinar la pestaña “Memory Analysis”, donde aunque probablemente no se muestre el objeto HttpSession como tal, podamos encontrar los tipos de objetos que estamos guardando en sesión. Para ello primeramente debemos de sacar la foto de los objetos en memoria previos al inicio de la navegación. Seguidamente usaremos la opción de “Start Tracking new objects”:

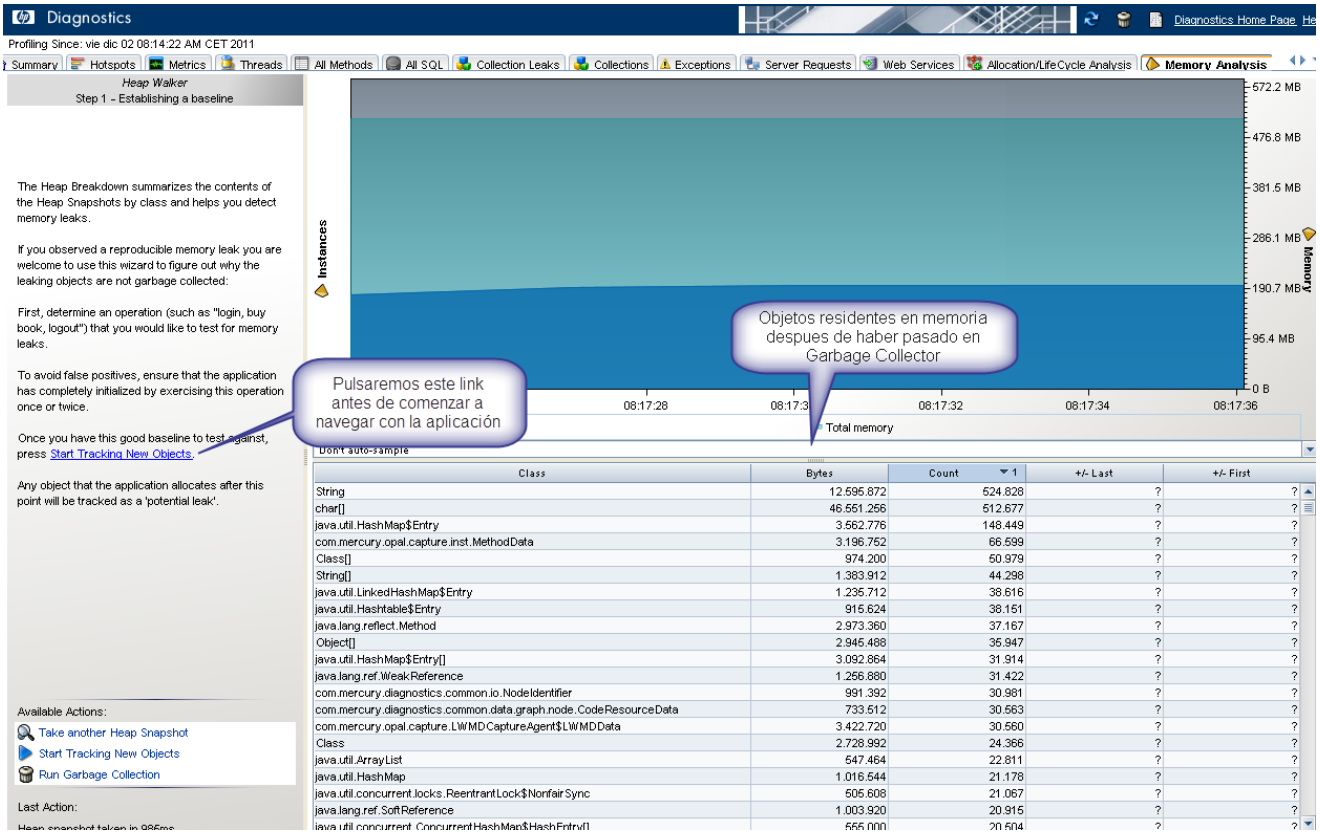


Figura 3.2– Memory analysis

El siguiente paso será acceder a la aplicación y navegar a través de las pantallas sospechosas de hacer el uso indebido del objeto sesión. Después de dicha navegación, nuestro gráfico de memoria habrá sufrido variaciones, incrementándose los objetos residentes en memoria:

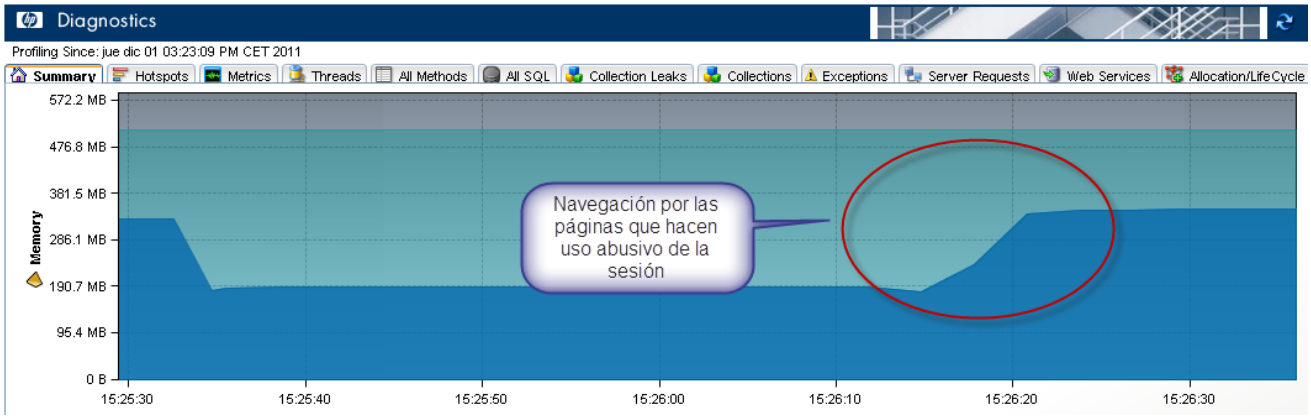


Figura 3.3– El uso de memoria aumenta con la navegación

La curva de uso de memoria se verá incrementada proporcionalmente al número de usuarios conectados. Recordar que desde la pestaña “Metrics” podemos acceder a los indicadores de Weblogic, donde entre otros, se encuentran el número de sesiones activas:

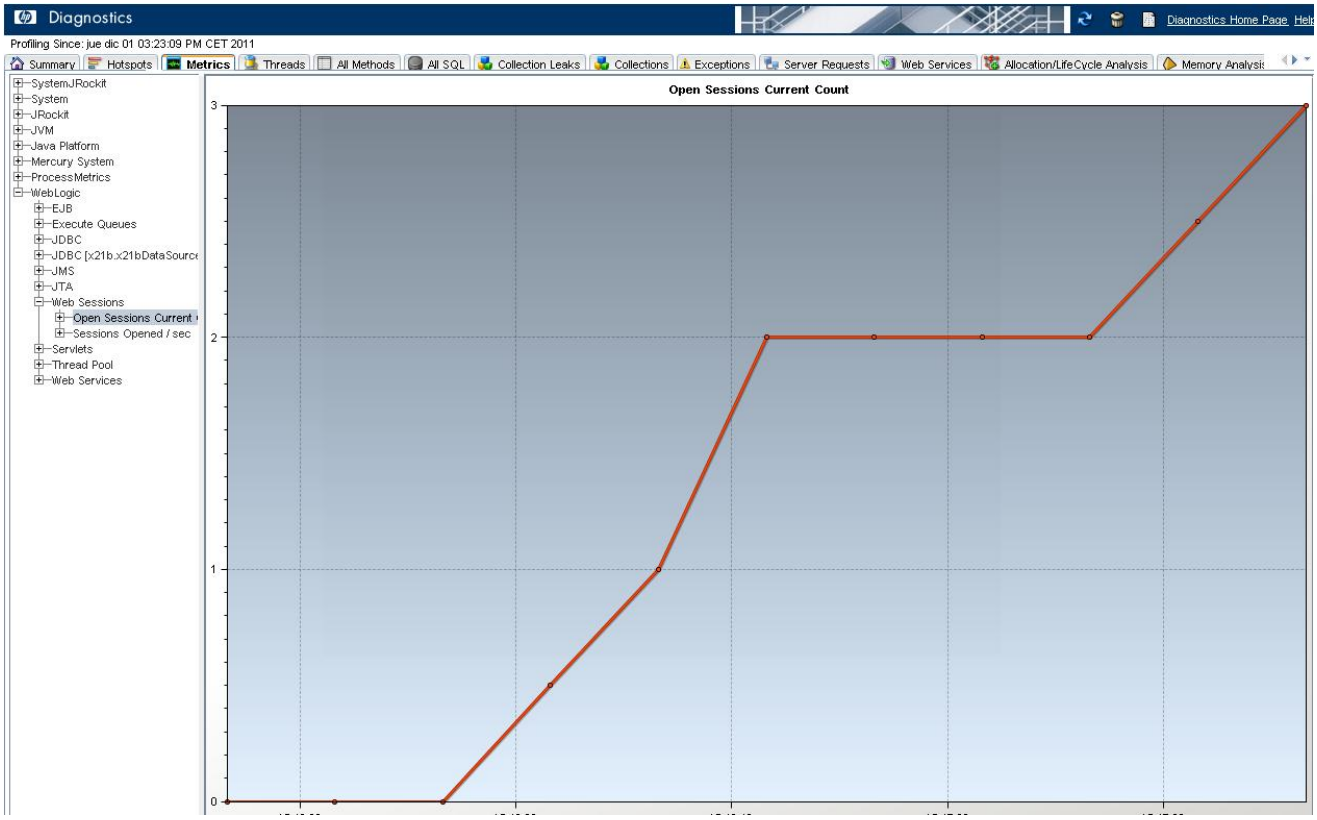


Figura 3.4– Número de sesiones activas

En el estado actual del caso práctico, todavía no podemos asegurar que el aumento de objetos de memoria ilustrado en la figura 3.2 se deba a la acumulación de objetos en sesión, ya que también podría venir provocado por la creación masiva de instancias de objetos que en teoría debería se capaz de eliminar el recolector de basura. Para descartar esta última casuística invocamos al Garbage Collector desde el HP Diagnostics:

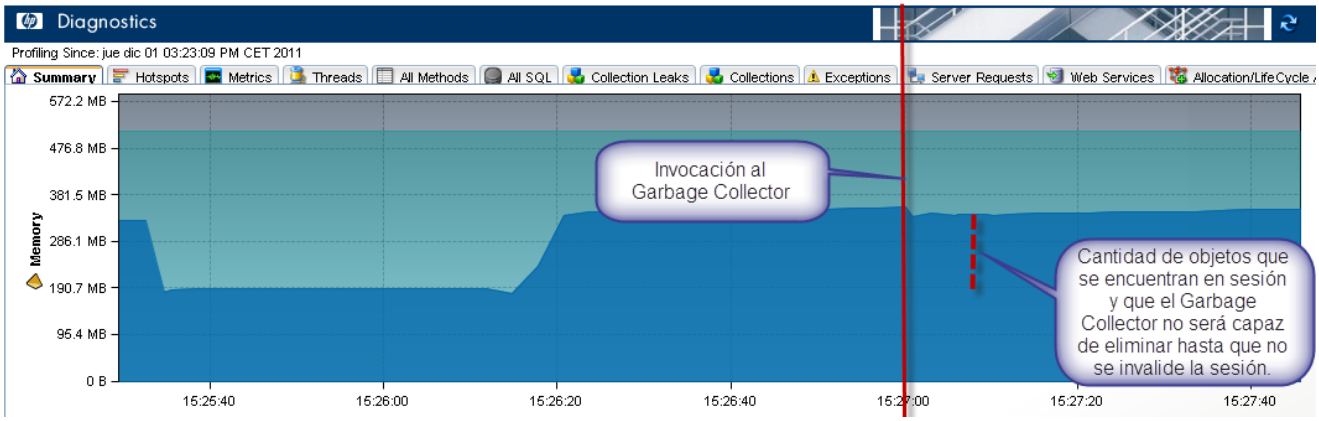


Figura 3.5– El Garbage Collector no es capaz de eliminar los objetos

Tal y como se representa en la figura 3.4, el Garbage Collector ha sido incapaz de eliminar los objetos de la memoria, ya estos se encuentran asociados a las sesiones del usuario aún activas.

Por otro lado podemos volver a la pestaña de “Memory Analysis” para tratar de identificar que objetos son los que el recolector de basura no ha sido capaz de eliminar. Para ello pulsamos sobre la opción “Stop Tracking Objects” y seguidamente hacemos clic sobre la opción “Capture object reference Graph” identificada por el icono . Una vez realizadas estas acciones podemos buscar en la tabla inferior los nuevos objetos creados desde que sacamos la primera foto de la memoria. Entre ellos podemos encontrar los objetos propios de la aplicación que se están quedando asociados a las sesiones de usuarios:

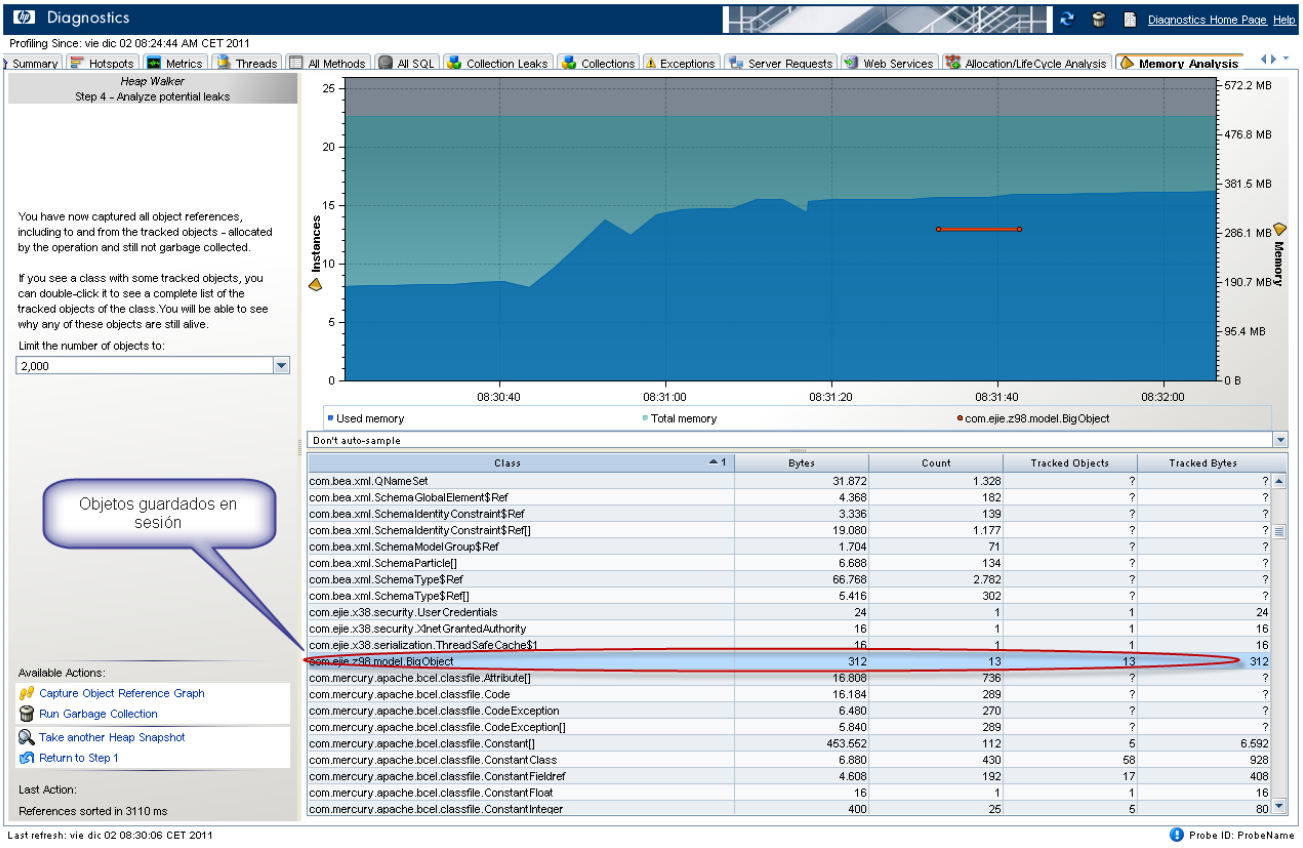


Figura 3.6– Identificar los objetos que no han sido eliminados por el GC

La afirmación de que los objetos que no han podido ser eliminados por el GC se encuentran en sesión puede ser un poco precipitada, ya que también podría tratarse de objetos asociados a variables estáticas. Para eliminar esta última sospecha, hemos creado un nuevo método en nuestro controlador (servlet) que invalida la sesión activa:

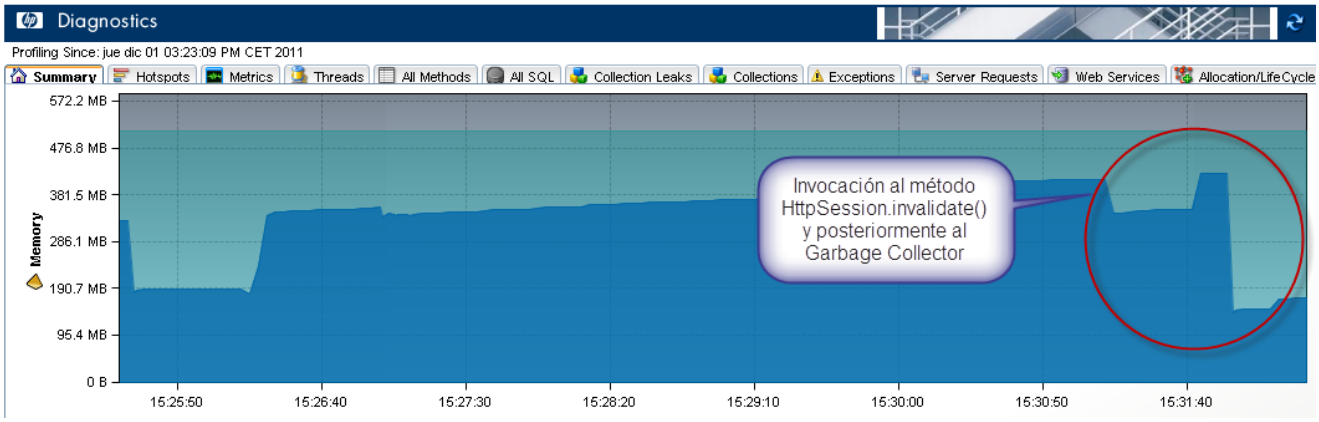


Figura 3.7– Invalidar sesión e invocar al Garbage Collector

Efectivamente, tal y como se muestra en la figura 3.7, al invalidar las sesiones activas e invocar al recolector de basura, la memoria vuelve aproximadamente a su estado inicial.

NOTA: En el anexo de este documento se presentan algunas técnicas, no relacionadas con HP Diagnostics, para monitorizar el tamaño de los objetos de sesión

3.2 Monitorizar las colecciones de objetos

Otro problemática generalizada en la construcción de aplicaciones J2EE es el uso abusivo de colecciones de objetos (HashMap, ArrayList, Collection, etc..). Generalmente estos tipos de objetos se utilizan como almacén de otros. Por ejemplo, para guardar los resultados de una consulta SQL. Con este tipo de objetos hay que trabajar con especial cautela, ya que si no los limpiamos correctamente pueden crecer en exceso provocando complicaciones en la memoria del servidor.

Hp Diagnostics incorpora la pestaña “Collections” donde se pueden estudiar el incremento de este tipo de colecciones

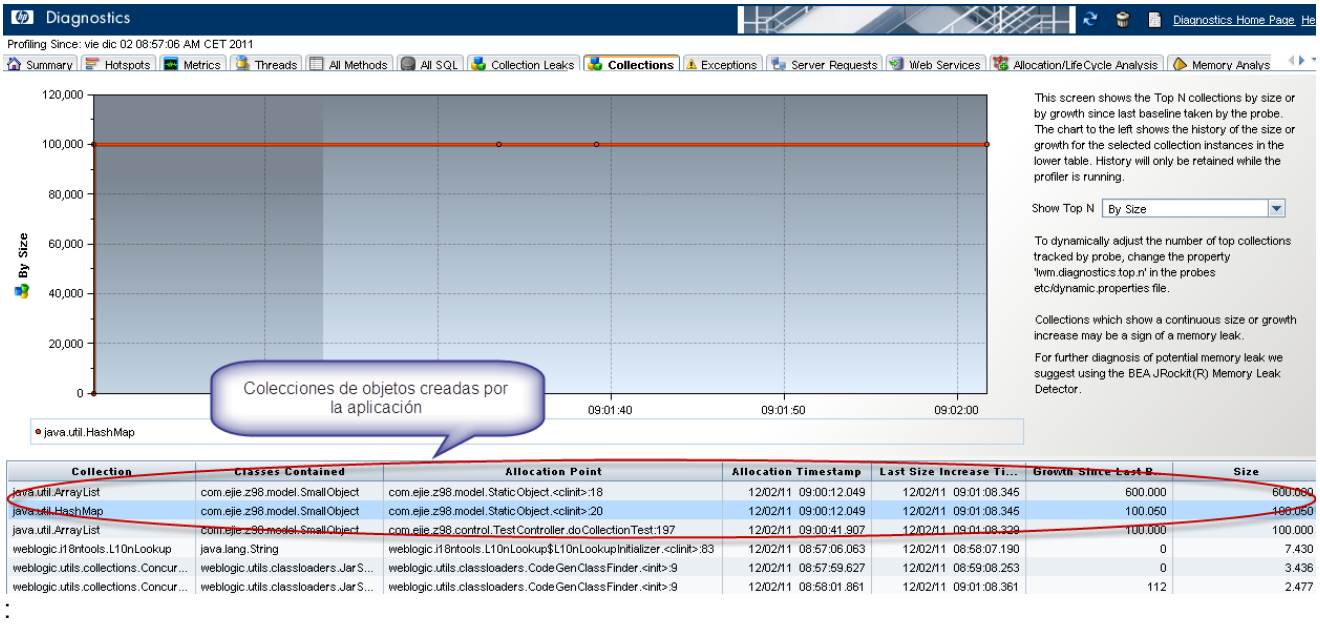


Figura 3.8– Colecciones de objetos creadas por la aplicación

3.3 Identificando un uso excesivo de la memoria del programa

Otro de los problemas frecuentes de las aplicaciones es la creación excesiva de objetos en memoria, originada principalmente por crear instancias dentro de bucles. Esto provoca que el uso de memoria del servidor de aplicaciones se incremente rápidamente y el recolector de basura se vea forzado a intervenir en numerosas ocasiones. Al tratarse de objetos creados dentro de un bucle, el GC será capaz de limpiar correctamente dichas instancias, aunque el hecho de estresar al recolector provoca un aumento de la latencia de todos los métodos ejecutados sobre la JVM, debido a que cuando entra en acción se detienen momentáneamente el resto de threads hasta que este termine su trabajo.

En estas ocasiones el gráfico de memoria de la pestaña “Summary” se caracterizará por picos constantes a lo largo de un corto periodo de tiempo. Los picos altos identifican a la ejecución del bucle que crea numerosas instancias de objetos y los pico bajos dan a entender que el GC ha entrado en acción limpiando la memoria:

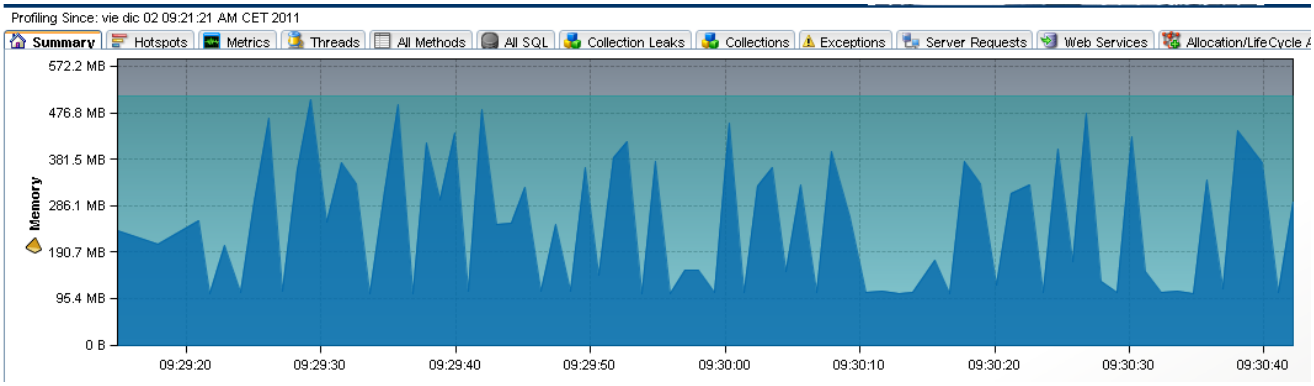


Figura 3.9– GC limpiando la memoria en numerosas ocasiones

Si viendo el gráfico anterior, aún nos queda la duda si se trata de la problemática aquí descrita, también podemos acceder a las métricas del GC para verificarlo:

Figura 3.10– Alta actividad del GC

En la figura anterior vemos como durante la ejecución del método problemático la actividad del recolector de basura es muy alta, para finalmente cuando termina su ejecución vuelve a su estado normal.

3.4 Identificando métodos excesivamente complejos

En ocasiones las aplicaciones J2EE se ven obligadas a implementar algoritmos complejos o recursivos que ralentizan las respuestas a las peticiones. En este apartado aprenderemos a identificarlas.

Si disponemos de un método con una lógica compleja, sin accesos a BD o a red, lo primero que podremos ver es que en la pestaña Summary se identifica como una petición con una lenta respuesta:

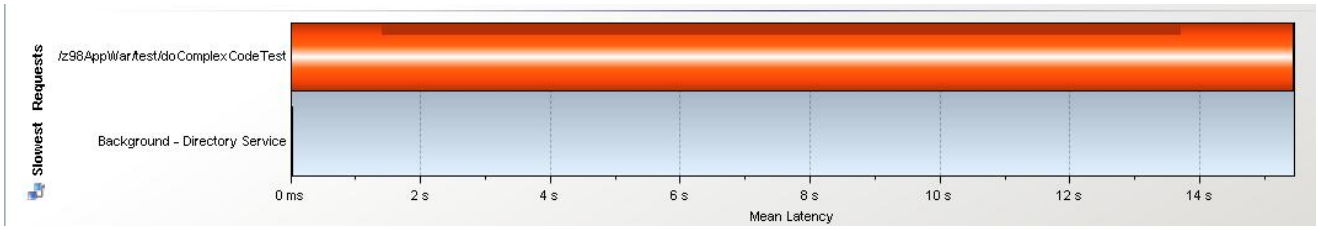


Figura 3.11– Petición identificada como lenta

Desde la pestaña Hotspots podemos ver los tiempos de CPU de la petición http y la latencia de los métodos asociados a dicha petición. En principio, al no tratarse de métodos de entrada/salida de disco o de red (nuestro caso solo ejecuta un algoritmo complejo), los tiempos de CPU deben ser similares a los de la latencia del método:

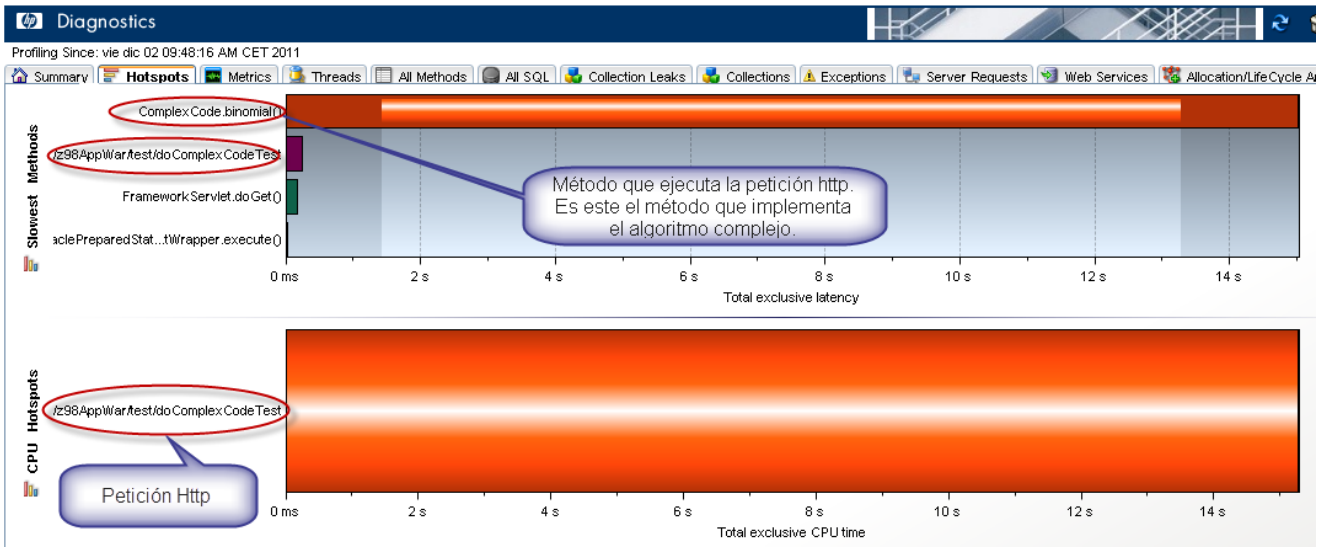
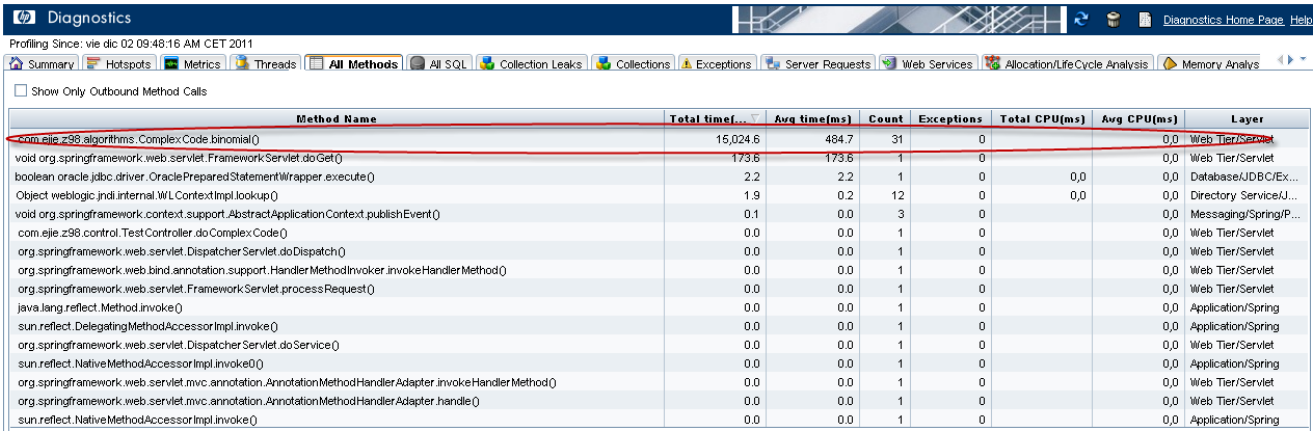


Figura 3.12– Vista desde la pestaña Hotspots

También podemos ver la latencia del método desde la pestaña All Methods:



Method Name	Total time(...	Avg time(ms)	Count	Exceptions	Total CPU(ms)	Avg CPU(ms)	Layer
com.ejie.z98.algorithms.ComplexCode.binomial()	15,024.6	484.7	31	0		0,0	Web Tier/Servlet
void org.springframework.web.servlet.FrameworkServlet.doGet()	173.6	173.6	1	0		0,0	Web Tier/Servlet
boolean oracle.jdbc.driver.OraclePreparedStatementWrapper.execute()	2.2	2.2	1	0	0,0	0,0	Database/JDBC/Ex...
Object weblogic.jndi.internal.WLContextImpl.lookup()	1.9	0.2	12	0	0,0	0,0	Directory Service/J...
void org.springframework.context.support.AbstractApplicationContext.publishEvent()	0.1	0.0	3	0		0,0	Messaging/Spring/P...
com.ejie.z98.control.TestController.doComplexCode()	0.0	0.0	1	0		0,0	Web Tier/Servlet
org.springframework.web.servlet.DispatcherServlet.doDispatch()	0.0	0.0	1	0		0,0	Web Tier/Servlet
org.springframework.web.bind.annotation.support.HandlerMethodInvoker.invokeHandlerMethod()	0.0	0.0	1	0		0,0	Web Tier/Servlet
org.springframework.web.servlet.FrameworkServlet.processRequest()	0.0	0.0	1	0		0,0	Web Tier/Servlet
java.lang.reflect.Method.invoke()	0.0	0.0	1	0		0,0	Application/Spring
sun.reflect.DelegatingMethodAccessorImpl.invoke()	0.0	0.0	1	0		0,0	Application/Spring
org.springframework.web.servlet.DispatcherServlet.doService()	0.0	0.0	1	0		0,0	Web Tier/Servlet
sun.reflect.NativeMethodAccessorImpl.invoke0()	0.0	0.0	1	0		0,0	Application/Spring
org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter.invokeHandlerMethod()	0.0	0.0	1	0		0,0	Web Tier/Servlet
org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter.handle()	0.0	0.0	1	0		0,0	Web Tier/Servlet
sun.reflect.NativeMethodAccessorImpl.invoke()	0.0	0.0	1	0		0,0	Application/Spring

Figura 3.13– Latencia del método

NOTA: La última imagen nos puede llevar a equivoco ya que el tiempo de CPU está vacío, aunque realmente equivale a la latencia total, ya que dicho método no tiene accesos a disco ni a red.

3.5 Procesamiento de XMLs

El procesamiento de XMLs suele ser una actividad costosa para la máquina virtual, y en ocasiones el origen de problemas de rendimiento de la JVM.

El caso práctico que aquí se presenta trata de comparar los dos modelos de procesamiento más comunes SAX y DOM. No profundizaremos en ellas tan solo haremos algunas observaciones a grandes rasgos:

- DOM: Consume mucha memoria pero a cambio es mucho más rápido si la JVM dispone de memoria suficiente. Ideal para XML de pequeñas dimensiones.
- SAX: Consume poca memoria ya que no carga todo el XML a procesar en memoria. Puede ser un poco más lento, pero es ideal para XML de grandes dimensiones.

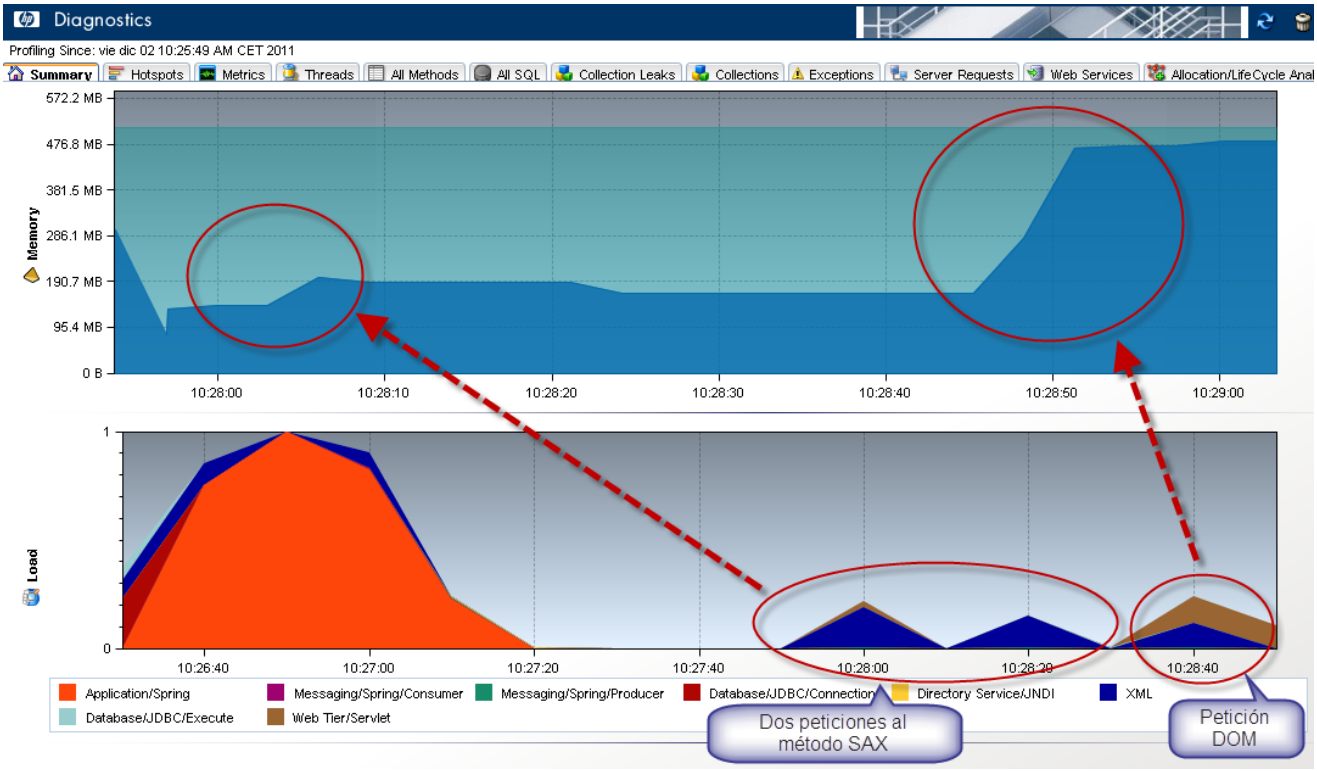


Figura 3.14– SAX y DOM

En la figura anterior se han realizado dos peticiones al método que procesa un XML de grandes dimensiones con SAX y otra petición con DOM. Podemos ver como HP Diagnostics ha identificado correctamente las capas XML y cómo el procesamiento de grandes XML con DOM consume mucha mas memoria que SAX.

Nota: Por defecto algunas métricas de XML se encuentran desactivadas. Es importante activarlas tal y como se refleja en el documento de instalación de HP Diagnostics.

Curiosamente al contrario de lo que se pueda pensar, en este caso la petición DOM ha tardado notablemente más que la petición SAX. Esto es debido a que el tener reservar tanta memoria también se traduce en un aumento de los tiempos de respuesta:

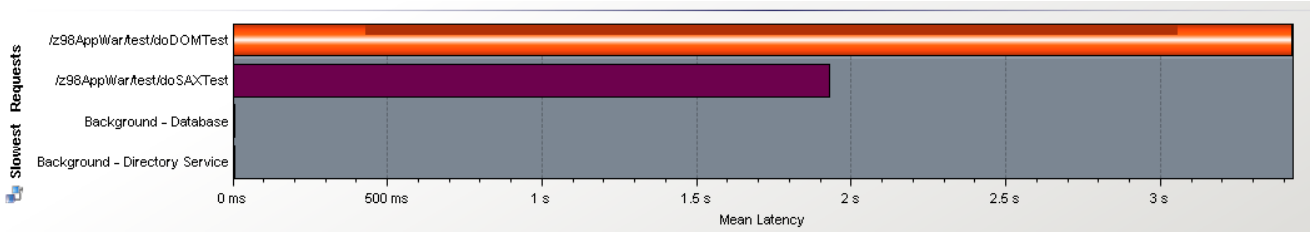


Figura 3.15– Comparativa de los tiempos de respuesta SAX y DOM

Para terminar con este caso de uso, modificamos el XML que leen ambos métodos haciéndolo notablemente más pequeño:

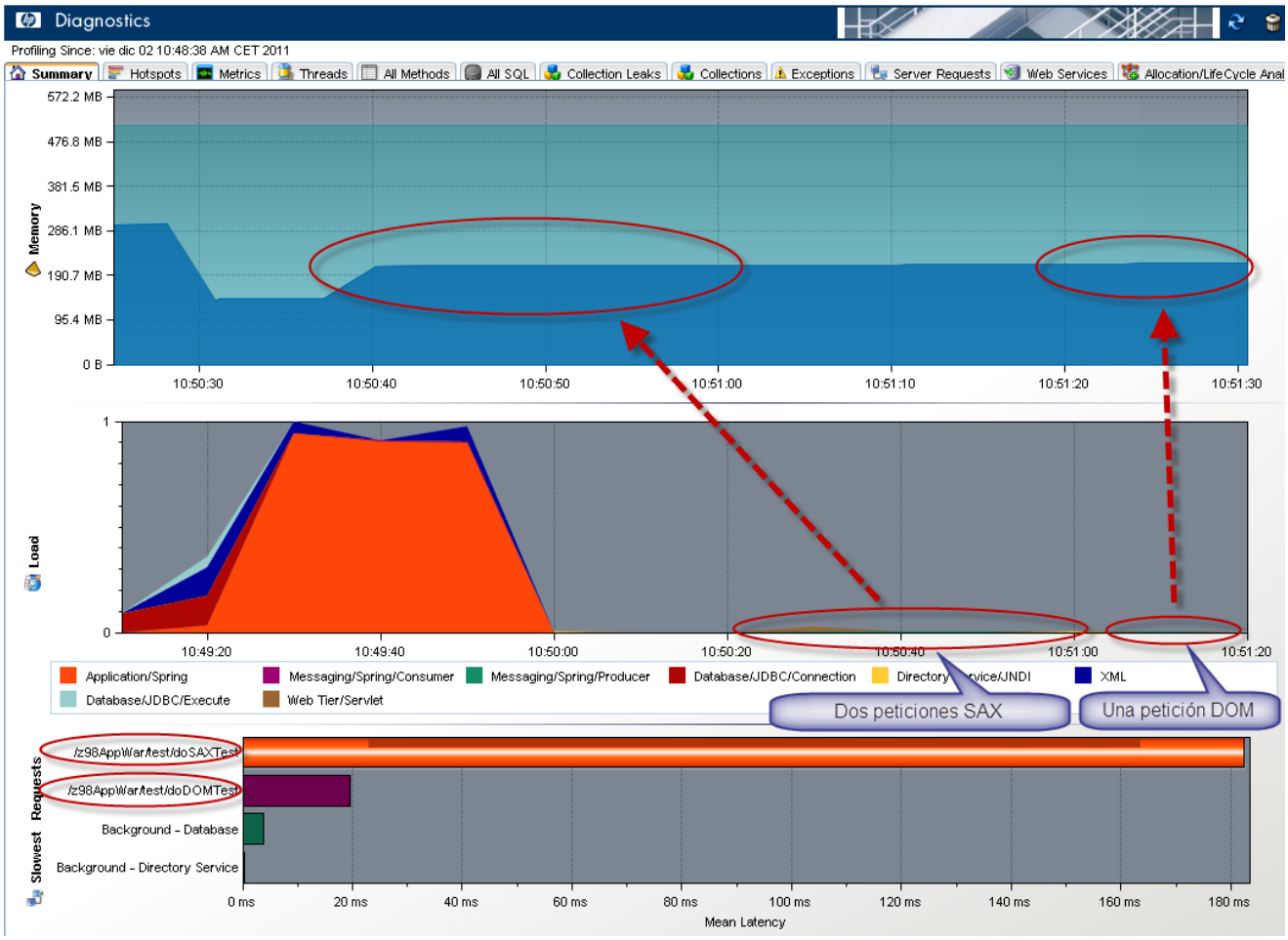


Figura 3.16– SAX y DOM procesando un XML de pequeñas dimensiones

Estudiando la figura 3.16 podemos obtener algunas conclusiones para el caso de procesamiento de XML de pequeñas dimensiones:

- Ambos consumen aproximadamente la misma cantidad de memoria.
- DOM es bastante más rápido que SAX.

3.6 Bloqueo de Threads (DeadLock)

DeadLock describe una situación en la que dos o más hilos se bloquean para siempre, esperando a los demás. Se trata de casos poco comunes, pero pueden ocasionar verdaderos quebraderos de cabeza para detectar sus causas.

Para el caso práctico de bloqueo de threads hemos tomado el ejemplo ilustrado en la documentación oficial de Oracle, incrustándolo en un controlador:

```
public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s"
                + "  has bowed to me!%n",
                this.name, bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s"
                + " has bowed back to me!%n",
                this.name, bower.getName());
        }
    }

    public static void main(String[] args) {
        final Friend alphonse =
            new Friend("Alphonse");
        final Friend gaston =
            new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
```

```
        }

}
```

(http://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html)

Cuando tratemos de acceder al controlador que bloquea los threads, no obtendremos ninguna respuesta, ya que el navegador se quedará infinitamente esperando una respuesta. Para descubrir que está ocurriendo, miraremos la pestaña Threads:

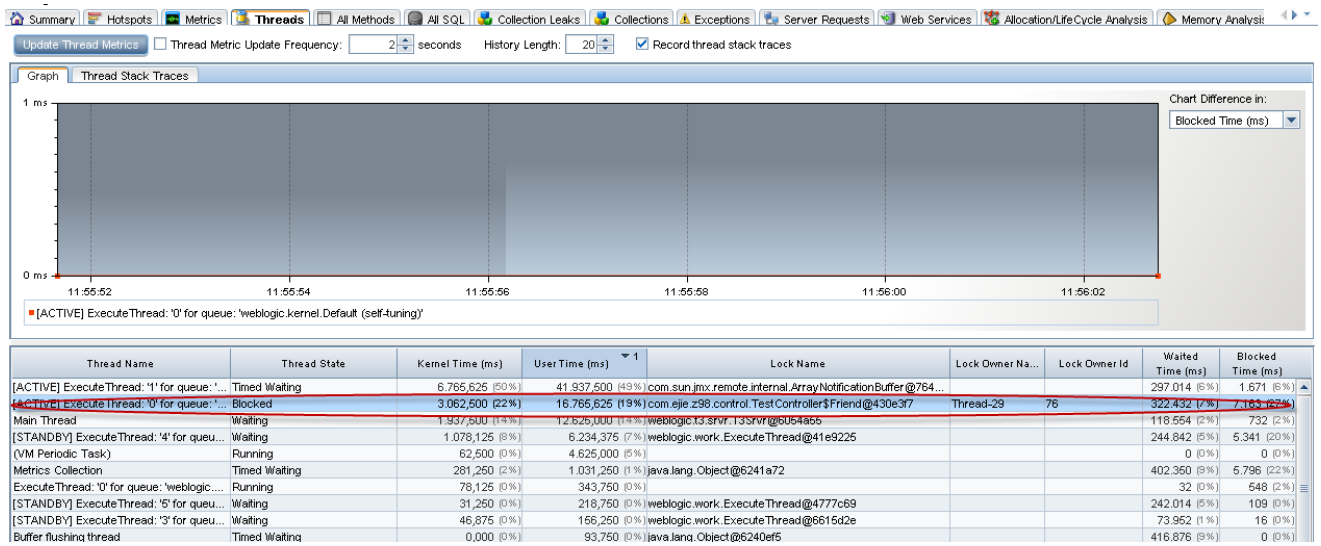


Figura 3.17– Thread bloqueado

En la imagen anterior podemos ver como se representa el thread como “Blocked”, asi como la clase responsable de este bloqueo (en este caso el controlador de la aplicación).

También podemos obtener mas datos de este thread bloqueado, realizando un volcado de threads a través de la subpestaña “Thread Stack Traces”:

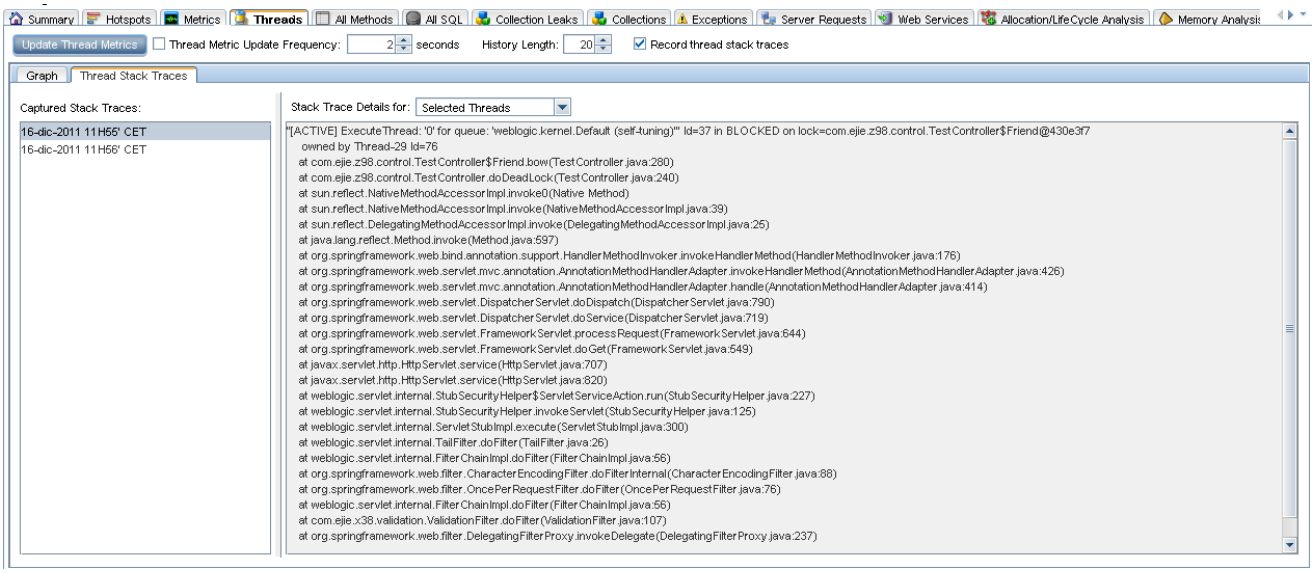


Figura 3.18– Stack Trace del Thread bloqueado

4 ANEXO 1: Monitorización de la sesión

En este anexo se presentan dos modelos de monitorización del uso del objeto sesión de una aplicación:

- A través del framework de diagnósticos de Weblogic 11
- A través de servlets empotrados en la aplicación.
- A través de JSPs empotradas en la aplicación.

4.1 Monitorización de la sesión a través de Weblogic Diagnostic Framework

Weblogic Diagnostic Framework (WDF) es una herramienta para la monitorización y diagnostico del servidor y sus aplicaciones. Entre la multitud de utilidades que incorpora en este documento nos centraremos en la monitorización de la sesión.

El primer paso será crear un nuevo módulo de diagnóstico desde la consola de administración. Para ello accedemos a la opción de Diagnósticos > Módulos de Diagnósticos:

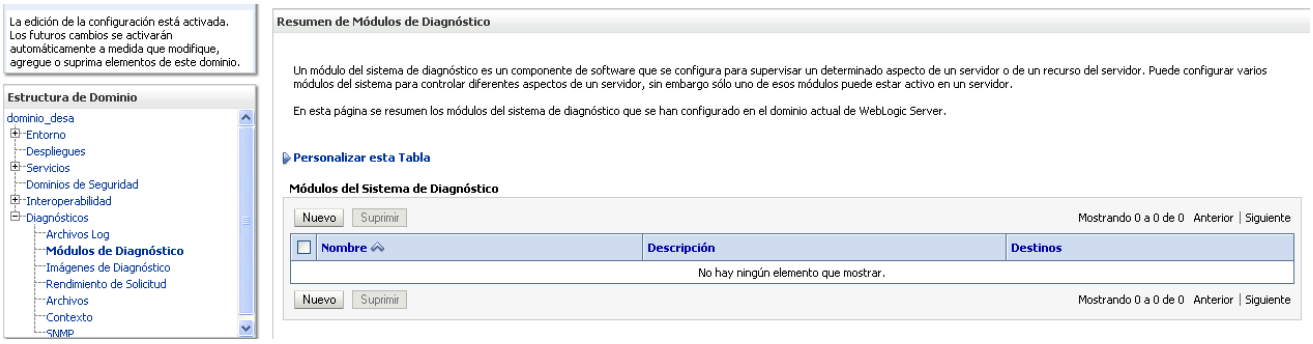


Figura 4.1– Módulos de diagnóstico

Pulsando el botón nuevo, crearemos el nuevo módulo.

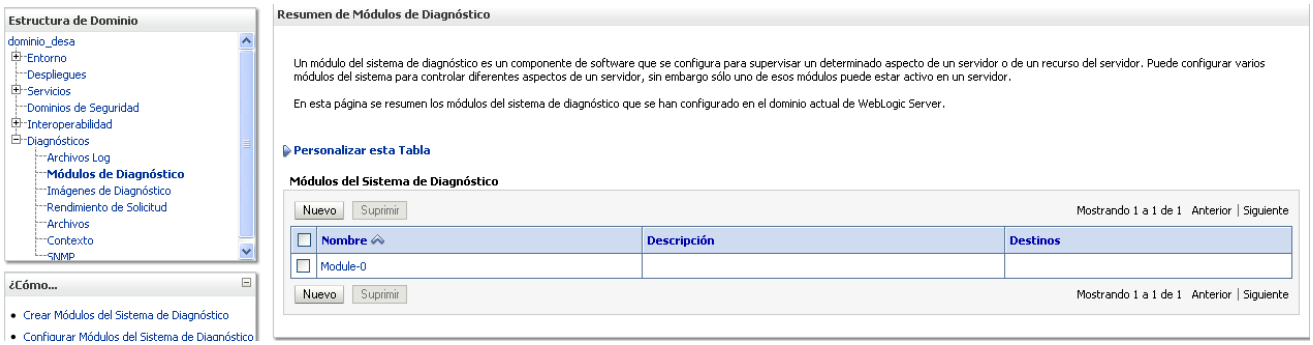


Figura 4.2– Nuevo módulo de diagnóstico

La siguiente tarea será habilitar dicho módulo y desplegarlo en el servidor, para ello hacemos clic sobre el nuevo módulo:

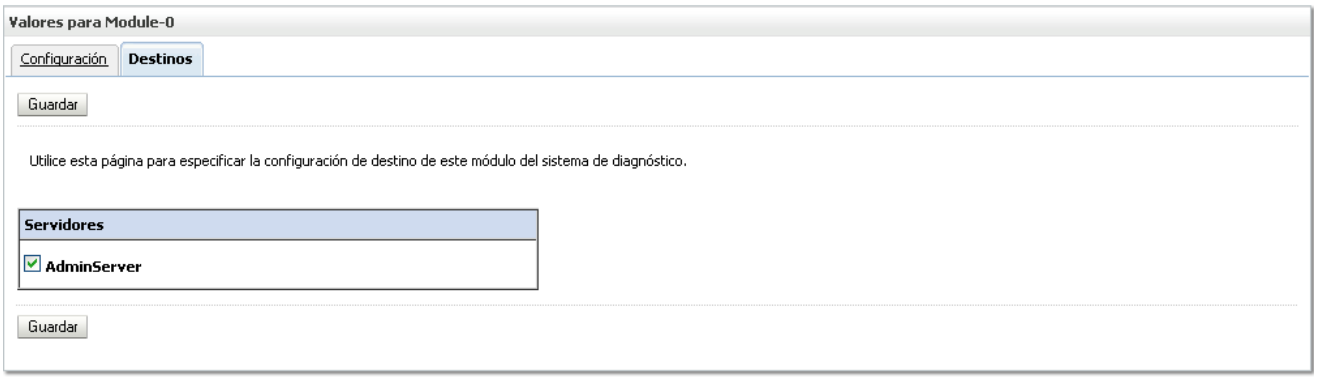


Figura 4.3– Despliegue en el servidor

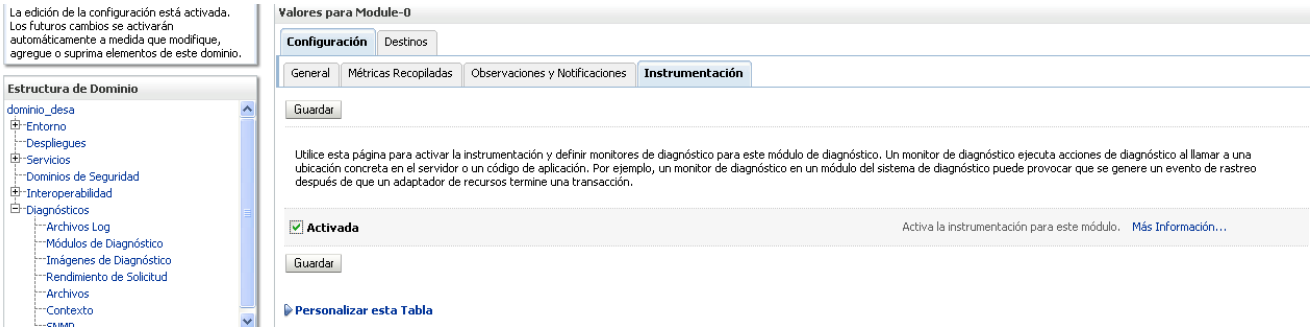


Figura 4.4– Activación del módulo de diagnóstico

Podemos verificar en el config.xml del dominio que hemos realizado correctamente los cambios. Si los pasos se han realizado correctamente, debería de aparecer la configuración del nuevo módulo:

```
<wldf-system-resource>
   <name>Module-0</name>
   <target>AdminServer</target>
   <descriptor-file-name>diagnostics/Module-0-3905.xml</descriptor-file-name>
   <description></description>
 </wldf-system-resource>
```

También podemos verficar el xml de la configuración del módulo:

```
<?xml version='1.0' encoding='UTF-8'?>
<wldf-resource….>
```

```
  <name>Module-0</name>
  <instrumentation>
    <enabled>true</enabled>
  </instrumentation>
</wldf-resource>
```

Una vez configurado el servidor, deberemos configurar nuestra aplicación para que envíe las métricas y se registren en la consola del servidor. Para ello creamos un nuevo fichero en el directorio META-INF del EAR llamado weblogic-diagnostics.xml, con el siguiente contenido:

```
<?xml version="1.0" encoding="UTF-8"?>
<wldf-resource xmlns="http://www.bea.com/ns/weblogic/90/diagnostics">
<instrumentation>
      <enabled>true</enabled>
      <wldf-instrumentation-monitor>
      <name>HttpSessionDebug</name>
      <enabled>true</enabled>
      </wldf-instrumentation-monitor>
      </instrumentation>
      </wldf-resource>
```

Una vez que reiniciemos el servidor, nuestra aplicación comenzará a enviar las métricas correspondientes a la sesión. Para verlas accederemos a la consola > diagnósticos > archivos log > EventsDataArchive:

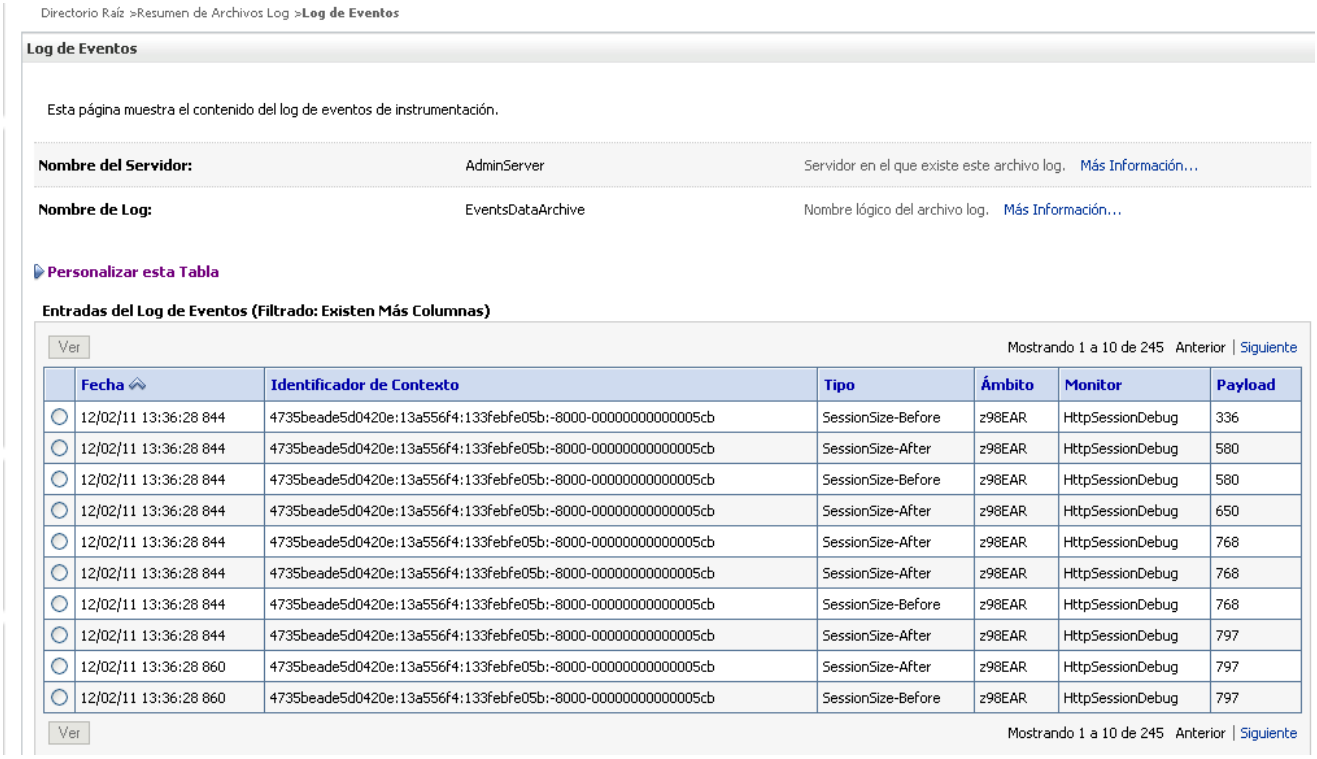


Figura 4.5– Log de sesión

4.2 Monitorización de la sesión a través de servlets

Existen mecanismos para acceder al objeto de sesión y averiguar sus características a través de un servlet desplegado en la propia aplicación. Sobre este concepto se basa un pequeño proyecto de software libre denominado MessAdmin (http://messadmin.sourceforge.net/).

Desde su página web podemos obtener las sencillas instrucciones de instalación. La vista de las características de la sesión sería similar a la siguiente:

Details for WebApp /svi (SCI Application)

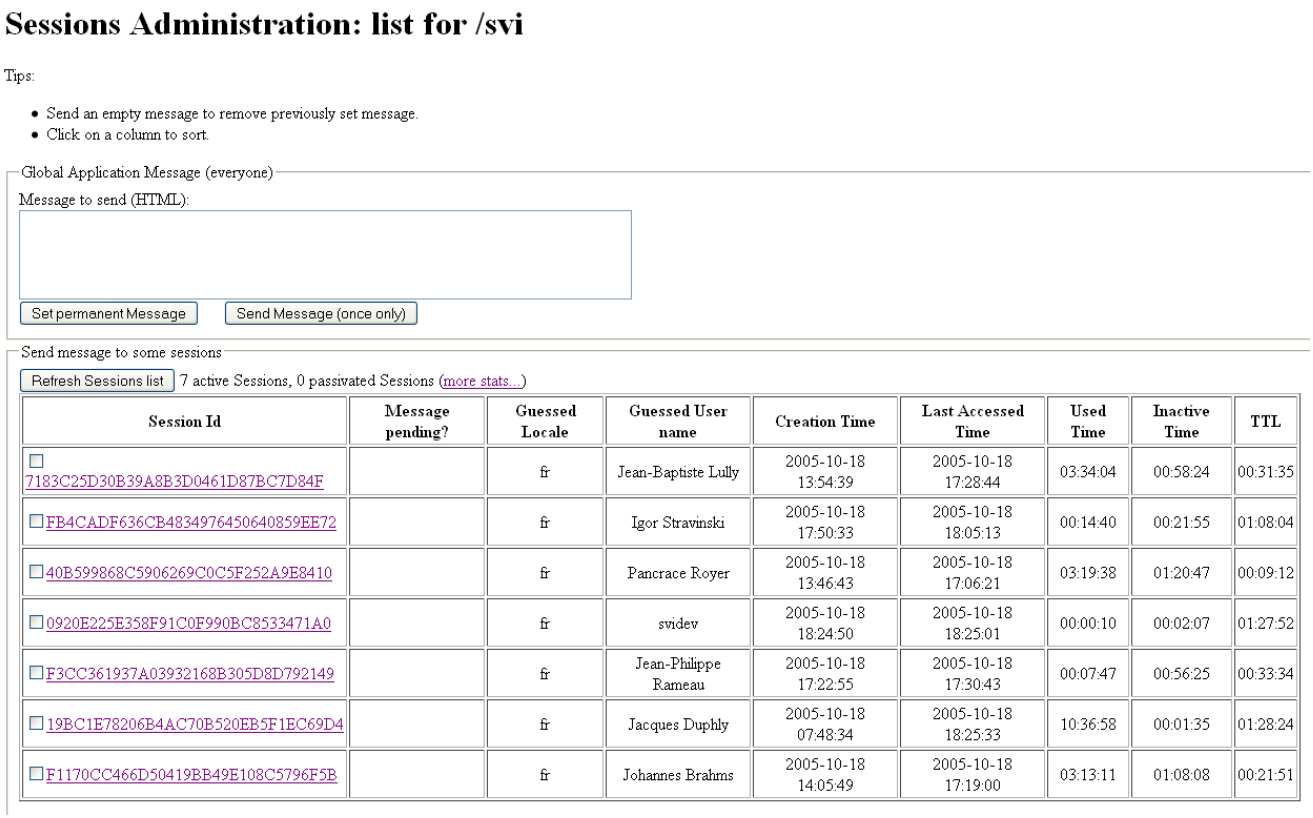
Platform	Apache Tomcat/5.x
Startup date	2005-10-18 03:30:12
Active sessions	7
Passive sessions	0
Maximum concurrent Sessions (peak)	8
at	2005-10-18 13:25:12
Total number of created Sessions (cumulative)	29
Total number of hits	27 781
Total Request size (bytes)	323 041
Total Response size (bytes)	118 807 888

Refresh

11 ATTRIBUTES

Remove Attribute	Attribute size (bytes)	Attribute name	Attribute value
Remove	151450	org.apache.struts.action.MESSAGE	org.apache.struts.util.PropertyMessageResources@1b86bf8
Remove	162	org.apache.struts.globals.MODULE_PREFIXES	[Ljava.lang.String;@16962b1
Remove	157	org.apache.struts.validator.STOP_ON_ERROR	true
Remove	397004	org.apache.commons.validator.VALIDATOR_RESOURCES	org.apache.commons.validator.ValidatorResources@1293656
Remove	8504948	org.apache.catalina.WELCOME_FILES	[Ljava.lang.String;@442f86
Remove	8512159	org.apache.struts.action.ACTION_SERVLET	com.svi.client.struts.SviActionServlet@cc295c
Remove	8512303	org.apache.struts.action.REQUEST_PROCESSOR	fr.improve.struts.taglib.layout.workflow.LayoutRequestProcessor@11603fa
Remove	8512533	org.apache.struts.action.PLUG_INS	[Lorg.apache.struts.action.PlugIn;@1c4b148
Remove	620556	org.apache.catalina.resources	org.apache.naming.resources.ProxyDirContext@8c11d5
Remove	126	database	svi
Remove	197153	org.apache.struts.action.MODULE	org.apache.struts.config.impl.ModuleConfigImpl@17cce3b

Figura 4.6– Vista de MissAdmin



Sessions Administration: list for /svi

Tips:

- Send an empty message to remove previously set message.
- Click on a column to sort.

Global Application Message (everyone)

Message to send (HTML):

Set permanent Message | Send Message (once only)

Send message to some sessions

Refresh Sessions list | 7 active Sessions, 0 passivated Sessions (more stats...)

Session Id	Message pending?	Guessed Locale	Guessed User name	Creation Time	Last Accessed Time	Used Time	Inactive Time	TTL
7183C25D30B39A8B3D0461D87BC7D84F		fr	Jean-Baptiste Lully	2005-10-18 13:54:39	2005-10-18 17:28:44	03:34:04	00:58:24	00:31:35
FB4CADF636CB4834976450640859EE72		fr	Igor Stravinski	2005-10-18 17:50:33	2005-10-18 18:05:13	00:14:40	00:21:55	01:08:04
40B599868C5906269C0C5F252A9E8410		fr	Pancrace Royer	2005-10-18 13:46:43	2005-10-18 17:06:21	03:19:38	01:20:47	00:09:12
0920E225E358F91C0F990BC8533471A0		fr	svidev	2005-10-18 18:24:50	2005-10-18 18:25:01	00:00:10	00:02:07	01:27:52
F3CC361937A03932168B305D8D792149		fr	Jean-Philippe Rameau	2005-10-18 17:22:55	2005-10-18 17:30:43	00:07:47	00:56:25	00:33:34
19BC1E78206B4AC70B520EB5F1EC69D4		fr	Jacques Duphly	2005-10-18 07:48:34	2005-10-18 18:25:33	10:36:58	00:01:35	01:28:24
F1170CC466D50419BB49E108C5796F5B		fr	Johannes Brahms	2005-10-18 14:05:49	2005-10-18 17:19:00	03:13:11	01:08:08	00:21:51

Figura 4.7– Vista de MissAdmin

4.3 Monitorización de la sesión a través de JSPs

Este mecanismo de monitorización es similar al anterior, pero en vez de utilizar Servlet, usaremos una JSP que empotraremos en nuestra aplicación:

```
<!doctype html public "-//w3c/dtd HTML 4.0//en">
<%@ page import="java.util.*"%>
<%@ page import="java.io.*"%>
<%@ page import="java.lang.reflect.*" %>

<html>
<head><title>Clusterable Session Object Test</title></head>
<body bgcolor="#FFFFFF">
<p>
<font face="Helvetica">

<h2>
<font color=#DB1260>
Check Clusterable Session Objects
</font>
```

```
</h2>

<p>This jsp tests whether the attributes held in the current HTTP session are
serialisable.
In order for web-applications to be capable of being replicated using WebLogic's in-memory
clustering,
all session attributes must be serialisable.</p>
<p>
<p>Therefore the object's class must implement the <font
face="Courier">java.io.Serializable</font> interface.
However, simply declaring that a class implements this interface is not sufficient and can
often lead
to problems when trying to cluster a web application for the first time. This JSP can be
used to help
diagnose such problems more quickly.</p>
<p>
<p>In order to guarantee that the session is truely serialisable this page tests each
session attribute
 by actually attempting to serialise it.
</p>

</p>



<br>

<%
       boolean clusterable= true;
       int attrNum= 0;
       ByteArrayOutputStream baos = null;
       ObjectOutputStream oos = null;
       byte[] array = null;
       ByteArrayInputStream bais = null;
       ObjectInputStream ois = null;
       Object deserializedObj = null;
       String attrName= null;
       Object attrValue= null;
       String impSerStr= null;
       String doesSerStr= null;
       String pctor = null;
    int size = 0;
    int totalSize = 0;


       Enumeration attrs = pageContext.getAttributeNamesInScope
(pageContext.SESSION_SCOPE);
%>
<center>
<table border=1 cellspacing=2 cellpadding=5 bgcolor=#EEEEEE>
<tr>
<th>#</th>
<th>Attribute Name</th>
<th>Class Name</th>
<th>Serializable</th>
<th>Serialises Okay</th>
<th>No args C'TOR</th>
<th>Bytes</th>
</tr>
<%
```

```
while (attrs.hasMoreElements())
{
       attrName= (String)attrs.nextElement();
    attrValue= pageContext.getAttribute (attrName, pageContext.SESSION_SCOPE);
    if (attrValue instanceof Serializable)
    {
       impSerStr= "Yes";
       doesSerStr= "Yes";

        pctor="<font color=\"red\"><b>No</b></font>";

       try
              {
                           baos = new ByteArrayOutputStream (2048);
                           oos = new ObjectOutputStream (baos);
                           oos.writeObject (attrValue);
                           oos.flush();
                           array = baos.toByteArray();
                           bais = new ByteArrayInputStream (array);
                           ois = new ObjectInputStream (bais);
                           deserializedObj = ois.readObject();

                           /* Public contstructor check */
                                 Constructor[] ctors =
deserializedObj.getClass().getConstructors();
                                 for (int i=0; i< ctors.length; i++) {
                                   if ( ( (Constructor)
ctors[i]).getParameterTypes().length == 0) {
                                        pctor="Yes";
                                   }
                                 }

                                 if (!pctor.equalsIgnoreCase("Yes")) {
                                   clusterable=false;
                                 }

                           size = array.length;
                           totalSize += size;
             }
             catch (NotSerializableException nse)
             {
                    doesSerStr= "<font color=\"red\">No </font><i>(See stack trace on
console)</i>";
                    clusterable= false;
                    application.log("serTest.jsp DIAG: Attribute #" + attrNum + " failed
to serialise: Name=" + attrName +
                           ", Class=" +  attrValue.getClass().getName() + " - " +
nse.getMessage() + " ...");
                    nse.printStackTrace();
             }
             catch (Exception e)
             {
                    doesSerStr= "<i><b>Error:</b> <font size=-1>" + e.getMessage()
+"</font></i>";
                    clusterable= false;
                    application.log("serTest.jsp DIAG: Attribute #" + attrNum + "
unexpected failure: Name=" + attrName +
                           ", Class=" +  attrValue.getClass().getName() + " - " +
e.getMessage());
                    e.printStackTrace();
```

```
                }
        }
        else
        {
           clusterable= false;
           impSerStr= "<font color=\"red\"><b>No</b></font>";
           doesSerStr= "<i>n/a</i>";
        }
%>
<tr>
<td><%=attrNum++ %></td>
<td><%=attrName%></td>
<td><%=attrValue.getClass().getName()%></td>
<td><%=impSerStr%></td>
<td><%=doesSerStr%></td>
<td><font size="-2"><%=pctor%></font></td>
<td><%=size%></td>
</tr>

<%
}
%>
</table>
<br>
<%
if (clusterable)
{
%>
<font size="+2" color="blue">Your HTTP session supports clustering</font>
<%
}
else
{
%>
<font size="+2" color="red">Your HTTP session does NOT support clustering</font>
<%
}

long mbConv = 1048576;

%>
<br>
<p>The total size of the objects in the HTTPSession is <%= totalSize %> bytes</p>

<h3>Runtime Data</h3>
<table border=1 cellspacing=2 cellpadding=5 bgcolor=#EEEEEE>
<tr><td align="left">The memory in use</td><td><%= ((Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory()) / mbConv) %> Mb</td></tr>
<tr><td align="left">The current free memory</td><td><%= Runtime.getRuntime().freeMemory()
/ mbConv %> Mb</td></tr>
<tr><td align="left">The total memory currently allocated</td><td><%=
Runtime.getRuntime().totalMemory() / mbConv %> Mb</td></tr>
<tr><td align="left">The maximum memory available</td><td><%=
Runtime.getRuntime().maxMemory() / mbConv %> Mb</td></tr>
<tr><td align="left">CPU Count</td><td><%= Runtime.getRuntime().availableProcessors()
%></td></tr>
</table>
<hr>
</center>
<p>
```

```
<font size=-1>Copyright &copy; 2002 by BEA Professional Services. All Rights
Reserved.</font>
</body>

</html>
```

Si accedemos desde el navegador a esta JSP, obtendremos un resumen de las sesiones abiertas por la aplicación:

Check Clusterable Session Objects

Check Clusterable Session Objects

This jsp tests whether the attributes held in the current HTTP session are serialisable. In order for web-applications to be capable of being replicated using WebLogic's in-memory clustering, all session attributes must be serialisable.

Therefore the object's class must implement the `java.io.Serializable` interface. However, simply declaring that a class implements this interface is not sufficient and can often lead to problems when trying to cluster a web application for the first time. This JSP can be used to help diagnose such problems more quickly.

In order to guarantee that the session is truely serialisable this page tests each session attribute by actually attempting to serialise it.

#	Attribute Name	Class Name	Serializable	Serialises Okay	No args C'TOR	Bytes
0	Position	java.lang.String	Yes	Yes	Yes	17
1	udaVirgin	java.lang.Boolean	Yes	Yes	No	78
2	udaTimeStamp	java.lang.Long	Yes	Yes	No	113
3	UserName	java.lang.String	Yes	Yes	Yes	15
4	SPRING_SECURITY_CONTEXT	org.springframework.security.core.context.SecurityContextImpl	Yes	Yes	Yes	1201
5	UidSession	java.lang.String	Yes	Yes	Yes	20
6	1292769252738	com.ejie.z98.model.BigObject	No	*n/a*	Yes	20

Your HTTP session does NOT support clustering

The total size of the objects in the HTTPSession is 1444 bytes

Runtime Data

The memory in use	244 Mb
The current free memory	267 Mb
The total memory currently allocated	512 Mb
The maximum memory available	512 Mb
CPU Count	2

Figura 4.8– JSP para visualizar sesiones