

## CAPÍTULO 12: PRUEBAS

En este capítulo se realizarán una serie de pruebas a la aplicación que hemos desarrollado. Para ello se utilizará un software de distribución gratuita llamado **JMeter**. La primera parte del capítulo tratará de una introducción del anterior software. En ella explicaremos los distintos **componentes de JMeter** pero no entraremos en detalles secundarios. También explicaremos cómo se puede **realizar un Plan de Prueba** que será lo que testará la aplicación. En la segunda parte veremos las **pruebas realizadas** y sacaremos las **conclusiones**. Incluiremos unas gráficas proporcionadas por el mismo software que ilustraran el capítulo.

### 12.1.- Introducción a JMeter

**JMeter** es una herramienta Java dentro del proyecto *Jakarta*, que permite realizar **Pruebas de Rendimiento y Pruebas Funcionales sobre Aplicaciones Web**. Es una herramienta de carga para llevar a cabo simulaciones sobre cualquier recurso software.

Existen un gran número de herramientas para realizar pruebas, gratuitas (*JUnit*, *JWebUnit*) y de pago (*LoadRunner*), pero **JMeter** destaca por su versatilidad, estabilidad y por ser de uso gratuito.

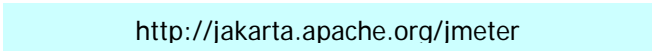
Inicialmente diseñada para pruebas de estrés en aplicaciones Web, hoy en día su arquitectura ha evolucionado no sólo para llevar acabo pruebas en componentes habilitados en Internet (HTTP), sino además en Bases de Datos, programas en Perl, solicitud FTP y prácticamente cualquier otro medio.

Además, posee la capacidad de realizar desde una solicitud sencilla hasta secuencias de requisiciones que permiten diagnosticar el comportamiento de una aplicación en condiciones de producción. En este sentido, simula todas las funcionalidades de un Navegador ("*Browser*"), o de cualquier otro cliente, siendo capaz de manipular resultados en una determinada solicitud y reutilizarlos para ser empleados en una nueva secuencia.

#### 12.1.1.- Instalación de JMeter

Para instalar el JMeter hay que tener instalado previamente el *JKD* de la plataforma correspondiente.

Para hacernos con el JMeter lo podemos conseguir de forma gratuita en la página principal de *Jakarta*:



<http://jakarta.apache.org/jmeter>

Figura 12. 1: Página descarga JMeter

Para instalarlo solo hay que descomprimirlo en el archivo que deseemos.

Cuando hayamos descomprimido el archivo, aparecerá una carpeta que se contendrá los siguientes elementos:

- **bin**

Este directorio contiene los ejecutables utilizados por JMeter, tanto para ambientes Linux (jmeter) así como Windows (jmeter.bat).

- **docs**

Contiene documentación acerca de JMeter

- **printable\_docs**

Contiene documentación en modalidad de impresión, acerca de JMeter

- **lib**

Este directorio contiene los archivos JAR empleados por JMeter requeridos en cualquier modalidad.

La ejecución de JMeter se lleva a cabo desde el directorio **bin**, que contiene las siguientes opciones:

- **jmeter | jmeter.bat**

Corresponde a los ejecutables de la interfaz principal para plataformas Unix y Windows, respectivamente.

- **jmeter-server | jmeter-server.bat**

Representan los ejecutables para el emulador de Servidor JMeter para plataformas Linux y Windows, respectivamente.

- **jmeter.properties**

Contiene propiedades de arranque para JMeter que son utilizadas por cualquiera de sus ejecutables.

- **jmeter.log**

Representa los registros (“logs”) generados al ejecutar JMeter.

- **users.xml | users.dtd**

Un archivo XML y su correspondiente DTD, empleados para definir características de usuarios que serán simulados por JMeter.

- **ApacheJMeter.jar**

Archivo JAR que contiene las principales clases de JMeter.

### 12.1.2.- Introducción a los Planes de Pruebas manuales

Una vez explicado por encima los componentes del programa pasamos a explicar con un ejemplo un Plan de Pruebas que es lo que nos permitirá testar una aplicación. Como explicación a este documento hay que señalar que depende la versión del programa que tengamos podemos tener los comandos en español o en inglés. Nosotros tenemos la versión en español por lo que pondremos los comandos en dicho idioma:

- Ejecutamos ***ApacheJMeter.jar*** ya que la vamos a hacer en un entorno Windows donde definiremos el Plan de Pruebas. Ponemos el nombre del plan de pruebas en la casilla correspondiente a los comentarios.

La parte izquierda representa la estructura o definición en árbol del Plan de Pruebas, mientras que en la derecha se nos muestra un *Form* que permite editar el elemento que tengamos seleccionado en la parte izquierda.

JMeter se basa en el concepto de “Elemento” (Element) y en una estructura en “Árbol” (Tree). Cualquier parte o rama del árbol puede ser guardada de forma independiente, para ser reutilizada en otras pruebas. Los elementos nos van a permitir configurar y definir el plan de pruebas:

- Elementos jerárquicos:
  - Listeners (elementos en escucha).
  - Config Elements (elementos de configuración).
  - Post-processors (post-procesadores).
  - Pre-processors (pre-procesadores).
  - Assertions (afirmaciones).
  - Timers (cronómetros).
- Elementos ordenados:
  - Controllers (controladores).
  - Samplers (agentes de pruebas).

Al crear un Plan de Pruebas se crea una “lista ordenada” de peticiones (request http) utilizando “Samplers” que representa los pasos a ejecutar en el plan. Normalmente, las peticiones se organizan dentro de “controllers”. Algunos tipos de “controllers” afectan al orden de ejecución de los elementos que controlan.

- Sobre el Plan, pulsando el botón derecho *Añadir->Grupo de hilos* creamos un grupo de hilos de ejecución para definir el número de usuarios a simular:

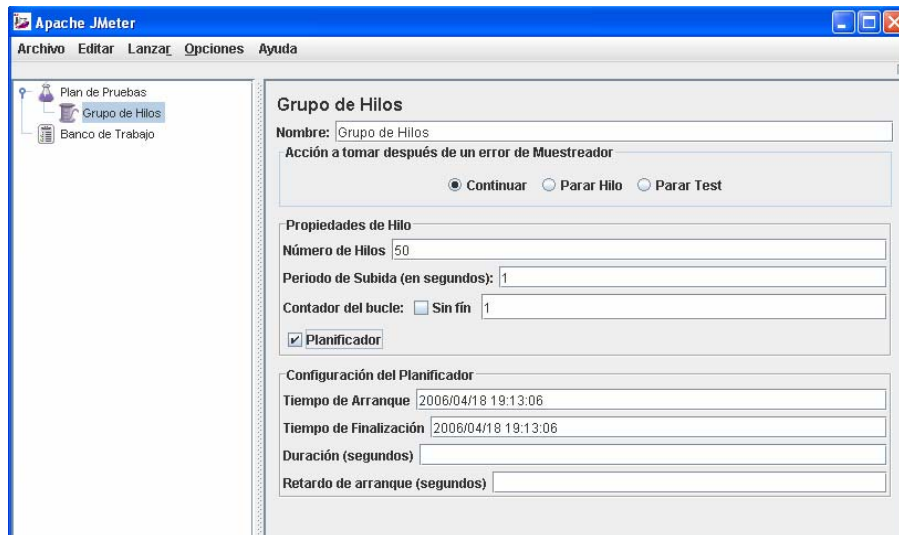


Figura 12. 2: Creación de un grupo de hilos

Podemos especificar el número de hilos en paralelo, así como el tiempo de arranque de cada uno, y un número de iteraciones que hará cada uno de ellos (puede marcarse como infinito). También podemos planificar (*scheduler*) la ejecución de la prueba indicando la hora de arranque y parada, o la duración del test en segundos y el tiempo de arranque del mismo. Otra cosa que debemos tener en cuenta es la acción a realizar en caso de error en la prueba por parte del hilo (continuar la ejecución, parar el hilo o parar todos los hilos de la prueba).

- JMeter posee dos tipos de controladores:

- **Samplers** (agentes de pruebas): Permiten enviar peticiones a un servidor. Por ejemplo "*http Request Sampler*" nos permite enviar peticiones http hacia un servidor. Los *samplers* se pueden configurar utilizando "*Configuration Elements*".
- **Logical Controllers** (controladores de lógica): Permite controlar el comportamiento de la prueba, y tomar decisiones en función de situaciones, etc. Por ejemplo, un controlador "*If Controller*" nos permite decidir si realizar o no un petición http en función de una condición. Cada controlador puede tener uno o más "*Default Elements*".

- JMeter permite crear "*Propiedades de la petición http por defecto*", para definir las propiedades por defecto que tendrán nuestras peticiones http representadas por el elemento "*Elementos petición http*". De esta forma cuando vayamos definiendo las distintas peticiones, no será necesario rellenar todos los campos de información ya que heredarán las propiedades por defecto definidas aquí. Por ejemplo, el nombre del servidor, el puerto, el protocolo, y algún valor que sea necesario siempre arrastrar en las peticiones.

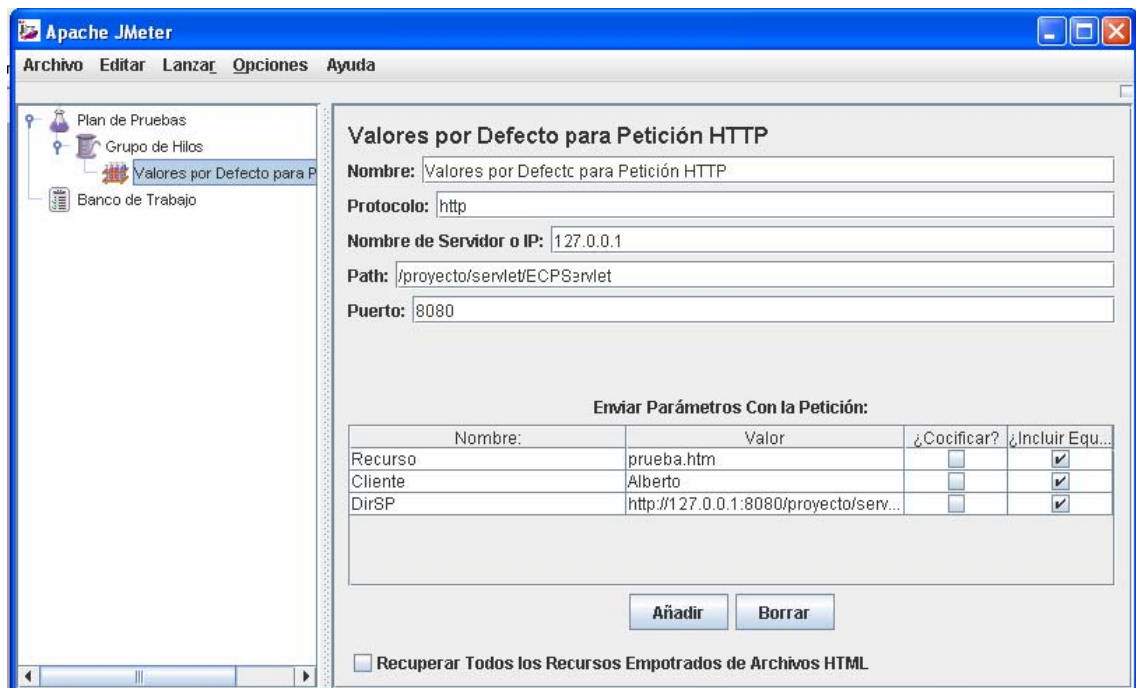


Figura 12. 3: Valores por defecto de las peticiones

- El siguiente paso es crear un controlador simple para agrupar una serie de peticiones, y le damos un nombre: *Añadir->Controlador lógico->Controlador simple*.

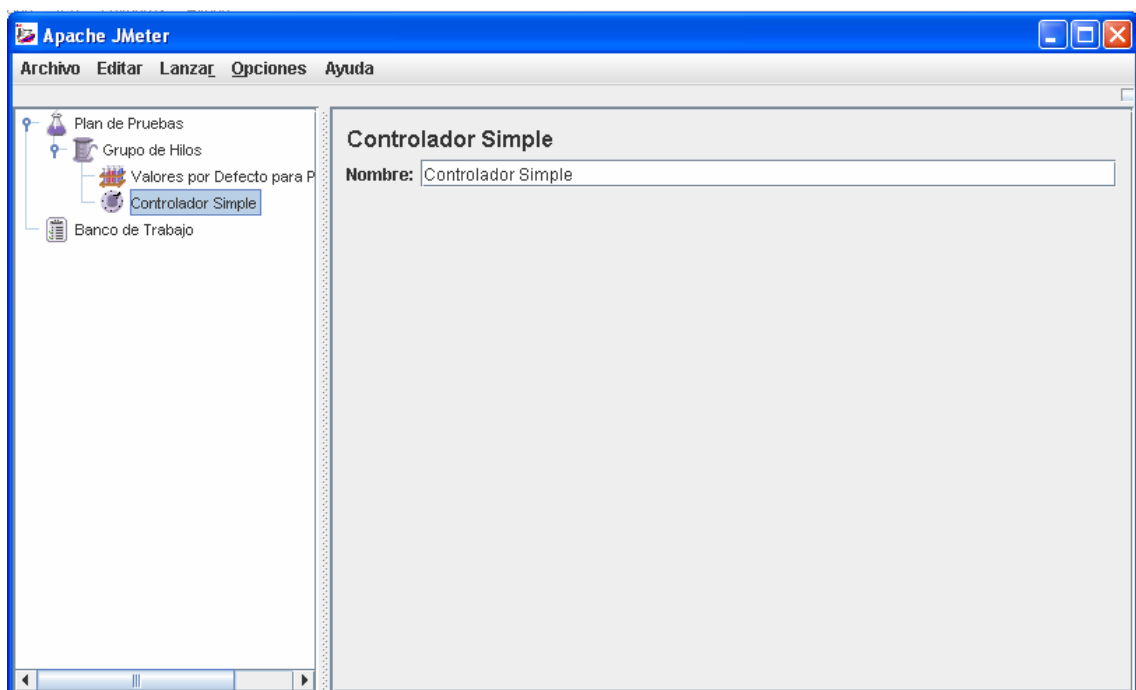


Figura 12. 4: Controlador simple

Y dentro de él, definimos una petición que le daremos el nombre de “Home”. Se hace mediante el comando: *Añadir ->Muestreador->Petición http*

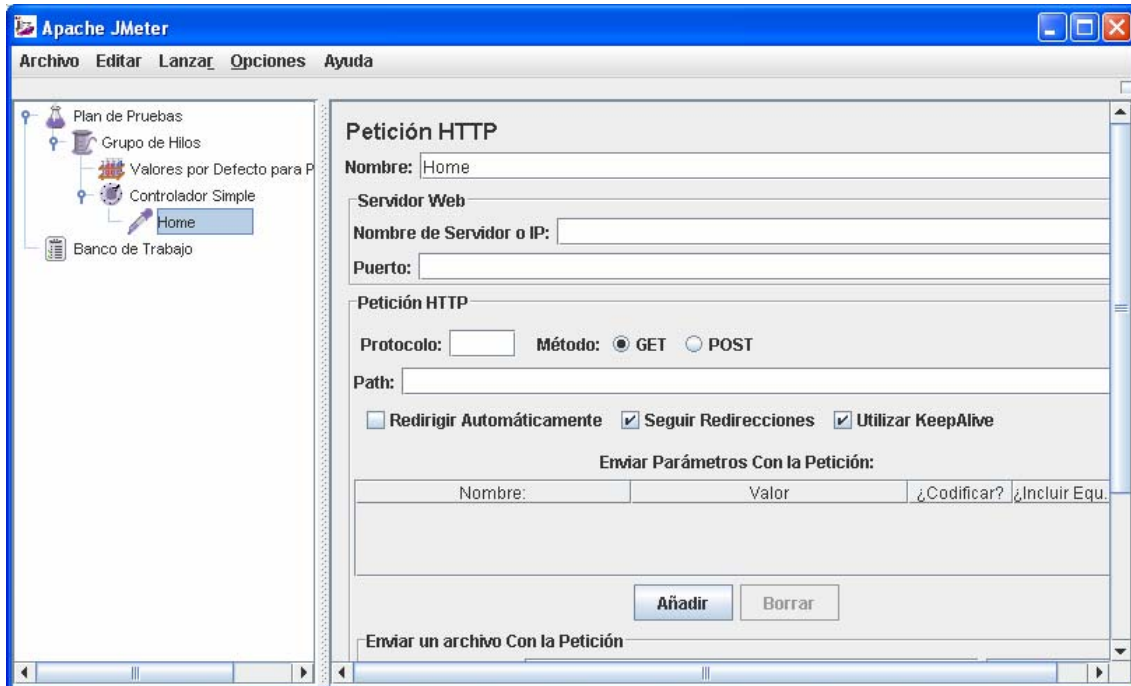


Figura 12. 5: Inclusión de una petición en el controlador simple

En esta pantalla, se nos permite controlar todos los parámetros de una petición http:

- Método: GET o POST.
- Path del recurso a pedir.
- Redirección automática.
- Seguir las redirecciones indicadas por el resultado de la petición.
- Uso de “KeepAlive”: Mantiene la conexión viva entre las distintas conexiones.
- Envío de parámetros en la petición.
- Envío de un fichero adjunto a la petición

Además los parámetros que no completemos serán heredados del elemento de configuración por defecto que nos precede en la jerarquía del árbol de definición del Plan de Pruebas.

- Añadimos una Aserción: Añadir->Aserción->Respuesta aserción.

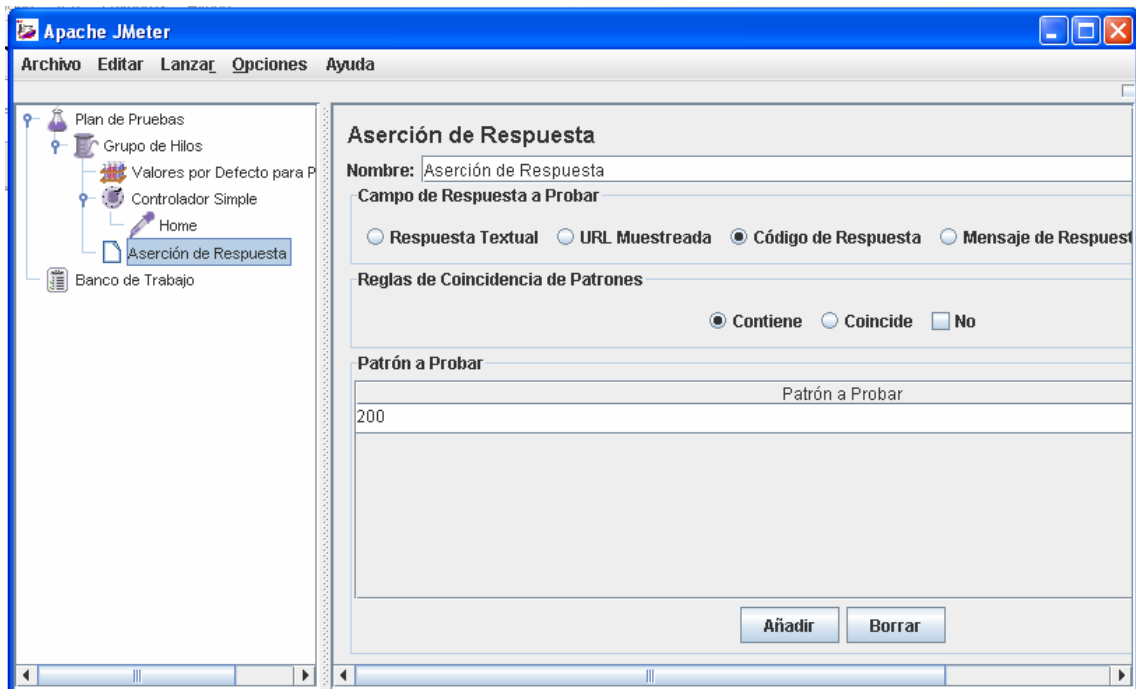


Figura 12. 6: Aserción de respuesta

Añadimos la condición de que el código de respuesta sea 200, correspondiente a una página servida correctamente. Se pueden añadir los siguientes tipos de afirmaciones como:

- Aserción de Respuesta, para comprobar la respuesta. Puede comprobarse el texto, o la URL, o el código de la respuesta, o el mensaje de respuesta, e indicar si coincide con una serie de patrones, o no.
- Aserción de Duración, para indicar un tiempo máximo de ejecución.
- Aserción HTML, para verificar que el HTML, XML o XHTML está correctamente construido (Utiliza Tiny).
- Aserción MD5Hex, para verificar que la suma MD5 es la especificada.
- Aserción de Tamaño, para verificar que el tamaño es menor, mayor o igual que uno especificado.
- Aserción XML, verifica que el resultado está bien formado en XML.
- Aserción de Beanshell, para programación de pequeños *shell scripts* que realizan verificaciones a medida.

También se pueden añadir preprocesadores que realicen acciones antes de enviar la petición.

- Contador: Para crear una variable contador, que puede ser referenciada en cualquier parte del test.
- Parámetros de Usuario: parámetros definidos por el usuario, que nos van a permitir definir una especie de constantes dentro del test.
- Parser de enlaces HTML: parsea la respuesta HTML del servidor, y extrae los links y forms.
- Máscara de Parámetro HTML.
- Modificador de re-escritura HTTP URL.
- Modificador de Parámetro de Usuario HTTP.

También se pueden añadir post-procesadores que realicen acciones después de enviar la petición:

- **Extractor de Expresiones Regulares:** Extrae cadenas de la respuesta (contenido o cabeceras) que cumplan una expresión regular.
- **Manejador de Acción para Status de Resultados:** Permite indicar la acción a tomar después de que se produzca un error. Tales acciones son continuar, parar el hilo, o parar el test.
- **Guardar Respuestas en un Fichero:** Permite almacenar en un fichero las respuestas obtenidas (todas o sólo las erróneas).
- **Generar Sumario de Resultados:** Resumen de información que se envía al *stdout* cada cierto tiempo (utilizado en modo *batch*).

Otra cosa que permite JMeter, es activar o desactivar una parte del Test, lo que es muy útil cuando se está desarrollando un Test largo, y se desea deshabilitar ciertas partes iniciales que sean muy pesadas o largas. También se pueden encadenar las peticiones que se deseen, y mover los elementos dentro del árbol.

- Finalmente, si vamos a lanzar el test de forma interactiva, necesitamos crear un *listener* dentro del grupo de hilos y obtener un informe. Los distintos tipos de informe son:

- **Resultados Aserción:** Muestra la URL de cada petición e indica los errores que se produzcan (aserciones que no se han cumplido) en el test.
- **Resultados de Grafico Completo:** simplemente muestra el tiempo.
- **Gráficos de Resultados:** Muestra un gráfico con los tiempos medios, desviación, throughput, etc. de la ejecución del Plan de Prueba.
- **Mailer Visualizar:** Permite enviar un e-mail si el plan de pruebas falla o no, o supera un determinado valor de fallos o éxitos.
- **Resultados del Monitor:** sólo para Tomcat 5.
- **Escritor de Datos Simple:** Vuelca los resultados en un fichero.
- **Visualizador Spline:** Gráfico de tiempo como Spline.
- **Informe Agregado:** Muestra una tabla con una fila por cada URL solicitada, indicando el tiempo mínimo, máximo, medio, etc.
- **Ver los Resultados en Tabla:** Muestra una tabla con todas las respuestas, la URL, tiempo y resultado de ejecución de cada una de ellas.
- **Ver los Resultados en Árbol:** Muestra un árbol con todas las respuestas y sus tiempos

- Por último nos queda guardar el Plan de Pruebas para un uso posterior (si queremos) y lanzar las pruebas. Si pulsamos *CTRL+R* o elegimos en el menú: *Lanzar->Arrancar*, empezaremos la ejecución de nuestro Plan de Pruebas. En ese momento se empezarán a ejecutar los hilos que configuramos en el grupo. En los diferentes informes que configuremos veremos los resultados.



### 12.1.3.- Generación automática del Plan de Pruebas

Otra forma que tiene JMeter de generar un caso de prueba es a través de una navegación de usuario. Para ello, indicamos que el JMeter actúe como Proxy, para capturar la navegación del caso de prueba. Se trata de configurar nuestro sistema para que el JMeter sea nuestro Proxy para que pueda almacenar los pasos que se dan cuando se navega. No se explicará con más detalle ya que utilizaremos la forma explicada de forma extensa para nuestra aplicación.

## 12.2.- Pruebas realizadas

Para testar nuestra aplicación es conveniente que la dividamos en dos partes bien diferenciadas, ya que dichas partes funcionan con tecnologías diferentes y tienen un análisis distinto:

- **Cliente J2ME:** Se realizarán pruebas de uso de memoria por parte del cliente con el fin de ver cuánto sobrecarga la aplicación al dispositivo. Se realizará con la herramienta “**Memory Monitor Extensión**” que proporciona el emulador **Wireless Toolkit**.
- **Aplicación SAML:** Para realizar las pruebas a esta aplicación se usará el software descrito en el primer apartado de este capítulo: **JMeter**. Puesto que esta parte es más compleja se realizará un número más extenso de pruebas.

### 12.2.1.- Prueba al cliente J2ME

El estudio del uso de memoria que hace la aplicación en tiempo de ejecución, consiste en realizar un análisis instantáneo y en tiempo de ejecución de las distintas tareas que se pueden realizar. Para ello haremos uso de herramienta “**Memory Monitor Extensión**” que proporciona el emulador **Wireless Toolkit**. Éste proporciona por un lado un gráfico en tiempo de ejecución del uso de memoria por parte de la aplicación, y por otro un listado detallado del uso que hacen de memoria los distintos objetos existentes en ella.

El proceso que vamos a realizar consiste en llevar a cabo todas las funcionalidades que ofrece la aplicación, observando para cada una de ellas el uso instantáneo que hacen de la memoria y anotando además cuales de ellas son las que más memoria utilizan. Por otro lado, además, este monitor de memoria permite observar una relación del uso de memoria que hacen los distintos objetos almacenados.

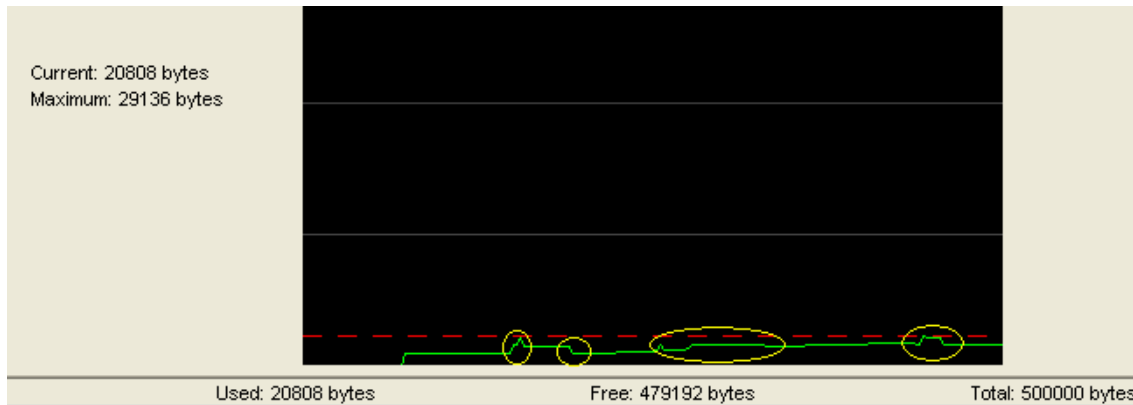


Figura 12. 7: Monitorización de la memoria consumida por el cliente

Como vemos en la figura hemos distinguido cuatro puntos que corresponden con pulsar los distintos comandos de nuestra aplicación:

- La primera marca corresponde con lanzar la aplicación, es decir, cuando desde la primera pantalla seleccionamos el comando **Launch**. Después del pico que marca este comando, llegamos a un valor de memoria de **18568 Bytes**.
- Acto seguido aparece un leve descenso de la ocupación de memoria y que hemos marcado como el segundo evento. Se corresponde con pulsar **Entrar** de la pantalla de entrada de la aplicación. Esto se debe porque se llama al recolector de basura para empezar una búsqueda sin nada en la memoria. Así tenemos un valor de ocupación de memoria de **12688 Bytes**.
- La zona que aparece ahora la relacionamos con lo que es nuestra aplicación. Es decir, en esa zona es cuando pedimos los recursos y los recibimos. Notamos dos picos leves. Uno se corresponde con la introducción de datos y el otro cuando recibimos los datos y los mostramos por pantalla. Alcanzamos una ocupación de memoria de **21652 Bytes**.
- En la cuarta zona vemos una pequeña bajada seguido de un pico de memoria. La acción que lleva asociada es una nueva búsqueda (de ahí la bajada) y una introducción incorrecta de los datos requeridos. El valor de ocupación es de **21564 Bytes**.

#### 12.2.1.1.- Conclusiones acerca del cliente J2ME

Mirando el resultado de la prueba podemos decir que hemos logrado crear un cliente que se ajuste a nuestros requerimientos. Esto se debe a su **baja carga de memoria**. En el gráfico vemos que tenemos una ocupación máxima de **29136 Bytes** de un máximo que tiene el dispositivo de **500000 Bytes**. Se puede ver de manera gráfica ya que tenemos nuestra aplicación muy abajo de lo que permite el dibujo. Este era uno de nuestros objetivos ya que si el cliente carga mucho el dispositivo, la aplicación no iría bien en aquellos que tengan muy limitada la memoria. De este modo cumplimos con el objetivo marcado de poderse utilizar en el mayor número de dispositivos.

También vemos que la aplicación **no tiene grandes variantes** en el uso de la memoria estando en unos márgenes relativamente pequeños. Indica que no hay grandes accesos o cargas que puedan provocar que nuestro sistema se quede bloqueado con lo que lo hacemos más robusto.

Estas características se deben a que es muy poca la dificultad del cliente creado. En él, básicamente, hacemos una conexión con una URL, le mandamos una petición y, lo que recibimos, lo mostramos por pantalla (siempre y cuando no nos manden un error).

### 12.2.2.- Pruebas a la aplicación SAML

Queremos demostrar mediante el software de prueba, la ventaja que introduce SAML a la hora de gestionar la información de autenticación. Hay que tener en cuenta que estamos probando la aplicación sólo, no la parte del cliente. Recordemos que SAML es especialmente útil cuando un usuario quiere acceder a varios sitios Web donde se les pide autenticación de identidad. El tiempo que gasta el usuario al introducir los datos es un tiempo nada despreciable que SAML elimina ya que se hace una primera autenticación y después esa información se transmite de forma transparente al usuario. Ese tiempo no entra en esta prueba ya que estamos testando la aplicación sólo sin preocuparnos del cliente. A los resultados habría que ponerle este tiempo para saber la potencialidad total de SAML.

#### 12.2.2.1.- Tiempo de respuesta

En esta prueba vemos el tiempo que tarda una petición en ser atendida, es decir, el tiempo que se tarda en responder a una determinada solicitud. Por razones obvias tratamos el tiempo medio de respuesta a las peticiones. **Esta prueba la enfocamos desde una comparación entre tener un contexto de seguridad permanente en el tiempo de prueba y otro que lo tenemos que establecer cada un segundo.** Para hacer esta prueba realizaremos tres experimentos sobre el mismo servidor que contiene a los tres servlet de la aplicación. En un caso real, los servlet estarán en máquinas distintas y por lo tanto, en distintos servidores. Los datos que se obtendrán no serán los de una aplicación puesta en máquinas independientes pero si se verá la respuesta según los distintos experimentos con lo que nos servirá para sus propósitos:

- Primero trataremos el tiempo de respuesta con una **carga baja del servidor**. Es decir, veremos cuánto tarda en responder cuando no existen demasiadas peticiones en el servidor.
- La segunda prueba será con una **carga media del servidor**. Aquí ponemos una carga al límite de la capacidad del servidor y veremos como es ahora el tiempo medido. Algunas de las muestras se perderá porque estamos trabajando en el límite.

- La última prueba será **sobrecargando el servidor** con más peticiones de las que puede servir. Veremos cómo se disparan las peticiones erróneas ya que algunas conexiones no se podrán establecer.

#### a) Carga Baja del servidor

En esta primera prueba vemos el tiempo de respuesta de una petición en el caso de estar con una carga del servidor baja. Para ello creamos dos grupos de hilos de 50 peticiones cada uno. En uno de ellos pondremos un cliente al que se le ha asociado un contexto de seguridad de validez permanente en el tiempo de prueba. Lo llamamos **Contexto Duradero**. El otro grupo creado es para un cliente con un contexto de autenticación de validez de un segundo que se llama **Contexto 1 segundo**.

Ponemos, en la definición del grupo de hilos del JMeter, un tiempo de subida de cuatro segundos que provocará que se tomen las peticiones casi de manera independiente ya que ese número de peticiones en cuatro segundos da un rendimiento bajo. De esta manera vemos como ante dos peticiones casi independientes, hay un tiempo de respuesta mayor para el contexto de un segundo que para aquel que es permanente. Esto es debido a que en el permanente no nos tenemos que conectar con el proveedor de identidad y con el de servicio sólo lo haremos una primera vez.

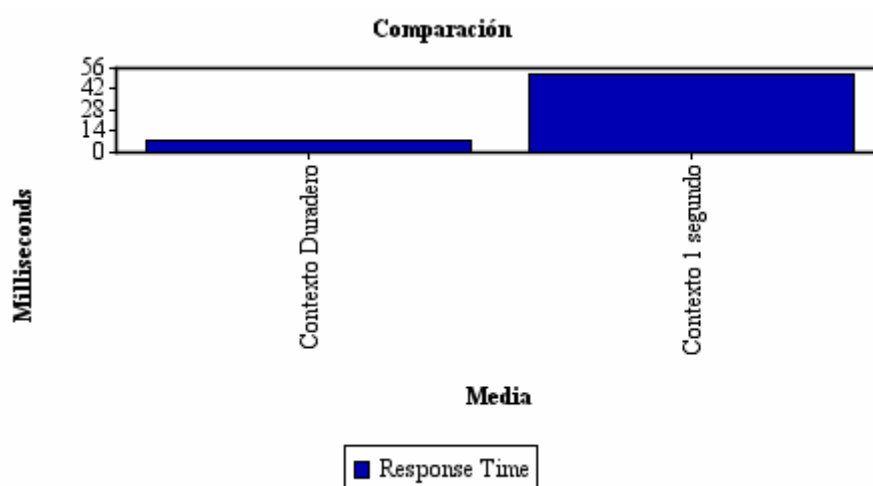


Figura 12. 8: Tiempo de respuesta con carga baja

Puesto que no hay demasiada carga, vemos que los tiempos de respuesta son muy pequeños. En otras palabras, vemos que nuestro sistema puede soportar esta carga sin problemas. Aún así vemos que la diferencia de tiempos entre ambos es muy grande por lo que el ahorro que introduce SAML empieza a ser importante.

URL	# Muestras	Media	Mediana	Linea de 90%	Mín	Máx	% Error	Rendimiento	Kb/sec
Contexto Dur...	50	8	0	20	0	200	0,00%	12,6/sec	3,61
Contexto 1 s...	50	51	10	260	10	471	0,00%	12,6/sec	3,63
TOTAL	100	29	10	30	0	471	0,00%	25,1/sec	7,20

Tabla 12. 1: Datos tiempo de respuesta con carga baja

Podemos comprobar que la premisa de bajo rendimiento se comprueba ya que da un valor total de **25,1 peticiones cada segundo** que es un valor de carga bajo. Al sistema le da tiempo de procesar las peticiones ahorrándonos cerca de un **85 %** del tiempo de respuesta para un contexto cada un segundo.

### b) Carga media del servidor

Ahora lo que haremos será trabajar en el límite de carga que puede soportar el servidor. Lo conseguimos aumentando el número de peticiones del grupo de hilos. Hemos probado unas trescientas peticiones para cada cliente en un tiempo de subida de cuatro segundos lo que son seiscientas peticiones en total.

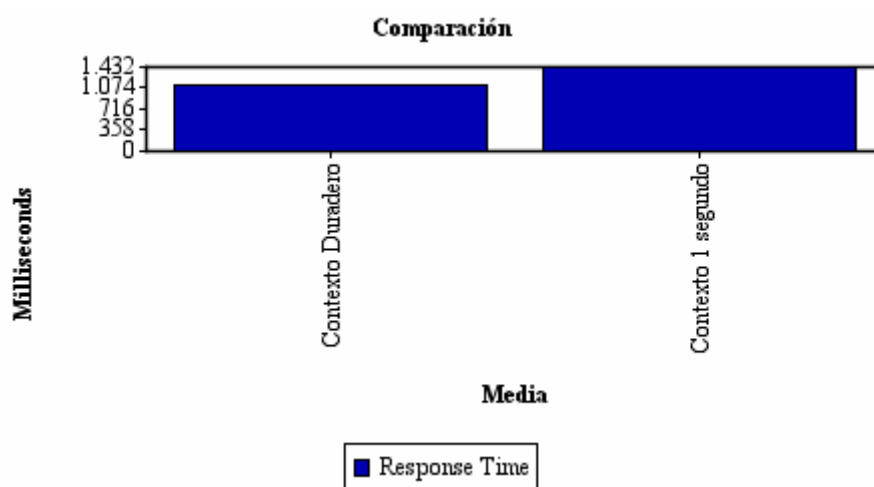


Figura 12. 9: Tiempo de respuesta con carga media

Todos estos datos lo sacamos de la tabla colocada más adelante. En ella vemos también que aparece un porcentaje de error de peticiones para el cliente del contexto de validez de un segundo. Esto se debe a que cuando no tenemos contexto de seguridad, tenemos que trabajar con un fichero. Al ser peticiones casi simultáneas, algunas entran en conflicto porque acceden a la vez al mismo fichero. Es un error que en la realidad no nos debe preocupar ya que no es lógico que un mismo cliente realice dos peticiones al mismo recurso de manera simultánea.

URL	# Muestras	Media	Mediana	Línea de 90%	Mín	Máx	% Error	Rendimiento	Kb/sec
Contexto Du...	300	1123	972	2193	40	2954	0,00%	46,6/sec	13,38
Contexto 1 s...	300	1427	1282	2754	130	3956	11,33%	40,8/sec	15,26
TOTAL	600	1275	1082	2313	40	3956	5,67%	78,5/sec	25,97

Tabla 12. 2: Datos tiempo de respuesta con carga media

Tenemos un rendimiento de mas de **78,5 peticiones cada segundo**, que consideramos una carga media para nuestro sistema. Es sistema empieza a cargarse por lo que el ahorro de tiempo de uno respecto al otro es ahora del **22 %**.

### c) Carga alta del servidor

Ahora sobrecargamos el servidor con el fin de comprobar cómo se comporta la aplicación ante esto. Ahora ponemos seiscientas peticiones para cada grupo de hilos dando un total de mil seiscientas peticiones.

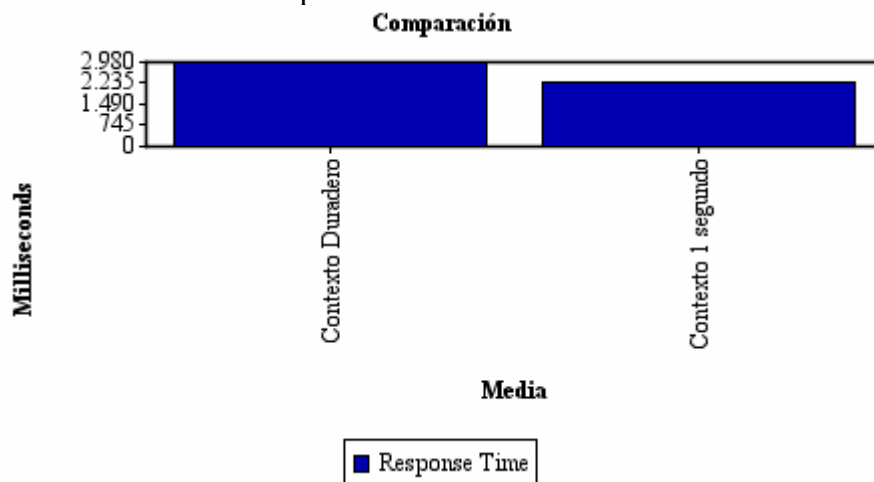


Figura 12. 10: Tiempo de respuesta con carga alta

Al ver la gráfica comprobamos que ocurre algo ilógico. ¿Cómo puede ser menor el tiempo de respuesta del contexto duradero mayor que el otro? La respuesta la vemos en la tabla de abajo. Está en los porcentajes de error que en el del contexto duradero es más alto. Esto se debe a que limitamos el número de hilos del servidor y muchas de las conexiones son rechazadas por el servidor. Esto es lo que provoca que haya un resultado ilógico.

URL	# Muestras	Media	Mediana	Linea de 90%	Mín	Máx	% Error	Rendimiento	Kb/sec
Contexto Du...	600	2977	991	9284	320	11516	68,67%	42,5/sec	39,98
Contexto 1 s...	600	2256	1002	7391	0	10354	64,33%	43,6/sec	39,65
TOTAL	1200	2617	1001	8562	0	11516	66,50%	84,7/sec	78,37

Tabla 12. 3: Datos tiempo de respuesta con carga alta

#### 12.2.2.2.- Conclusiones a las pruebas de la aplicación

Como comentamos, el **JMeter lo que realiza son pruebas de estrés**. De este modo lo que hemos realizado es cómo se comporta la aplicación en el medio que se desarrolla: un servidor de servlet. Vemos que el servidor es el factor que más limita a la aplicación. Ante pruebas de bajo rendimiento, pocas peticiones al segundo, el ahorro de tiempo al usar SAML con contexto de autenticación duradero es muy grande. Este ahorro disminuye conforme la carga del servidor se aumenta. Ya se comentó que esto era debido a dos factores:

- Al aumentar la carga, muchas peticiones intentan acceder al fichero de autenticación a la vez lo que provoca fallos. En realidad estas pérdidas en un entorno real no se producirían tanto ya que no es muy probable que el mismo cliente acceda a un mismo recurso con muchas peticiones al mismo tiempo.

- La otra limitación no proviene de nuestra aplicación sino del propio servidor. Este tiene limitado el número de hilos concurrentes a ciento cincuenta. Este parámetro es configurable pero siempre será un número finito por lo que siempre aparecerá esta limitación. En la realidad, cada servlet funcionará en servidores separados con lo que incrementa el rendimiento de peticiones que se pueden soportar a la vez. También podemos permitir que convivan más hilos a la vez. Incluyendo las dos medidas se puede aumentar de forma significativa el número de peticiones a las que el sistema puede dar servicio.

Como **conclusión final** hay que comentar que la inclusión de **SAML ahorra un tiempo de respuesta significativo** en el acceso a recursos por parte de un mismo cliente. Este era otro de los objetivos que pretendíamos conseguir incluyendo SAML en nuestra aplicación.