**Comparativa de implementaciones Long Pooling**

Germán Ramos García ([german.ramos.garcia@bbva.com](mailto:german.ramos.garcia@bbva.com)) 27/05/2013

# Introducción

A día de hoy, cada vez resulta más y más necesario la creación de aplicaciones web donde el servidor pueda transmitir datos inmediatamente a los clientes proactivamente sin esperar a que el cliente solicite dichos datos.

El sistema tradicional de consultas periódicas por parte del cliente (**Pooling**) presenta dos principales carencias:

* Los datos no llegan al cliente cuando el servidor quiere, sino que debe esperar a que transcurra el tiempo de refresco prefijado.
* Se realizan constantemente peticiones AJAX por parte del cliente y en la mayoría de los casos no hay nuevos datos que transmitir. Esto aumenta considerablemente el tráfico de red y además sobrecarga al servidor de peticiones innecesarias.

Una forma más eficiente de enviar datos puntuales desde el servidor al cliente es el sistema llamado **Long Pooling**. Este sistema consiste en que el cliente establece una conexión con el servidor y esta conexión permanece abierta hasta que el servidor transmite algún dato. Una vez recibido el dato por el cliente vuelve a establecer la conexión y permanece a la espera. Este sistema presenta las siguientes características:

* Permite al servidor realizar **notificaciones instantáneas** a los clientes
* Se reduce significativamente el tráfico de red
* Se reduce significativamente la sobrecarga del servidor cuando hay muchos clientes

Hasta hace poco, los frameworks web no tenian soporte para la implementación de sistemas de Long Pooling. Un servidor web tradicional no soporta más de 1000 o 2000 conexiones simultáneas antes de colapsar puesto que utiliza un hilo para cada conexión. Sin embargo, las cosas han cambiado notablemente y vamos a ver varias alternativas para hacer esto de una forma eficiente y con capacidad para manejar varios miles de clientes concurrentes de manera asíncrona.

# Escenario de pruebas

Las pruebas se han desarrollado sobre un servidor Ubuntu 12.04 Server en una máquina virtual de VirtualBox. La máquinaa virtual está configurada con 1 core y 6Gb Ram.

Para los clientes, se han creado 3 maquinas virtuales con 1 core y 256Mb Ram cada una.

El equipo hardware donde corre el hipervisor es un iMac con procesador Intel Core i7 3.4GHz 8GB RAM, sistema operativo MacOSX 10.8.2 y VirtualBox 4.2.12.

# 

# Objetivo, procedimiento y parámetros a valorar

El objetivo de la prueba es evaluar la carga del servidor, especialmente el uso de memoria, ante miles de conexiones concurrentes.

Para realizar la prueba se ha realizado, en diferentes lenguajes, el siguiente programa:

* Servidor web aceptando peticiones GET.
* Almacena la conexión en una lista.
* Cuando el servidor recibe la orden, responde a todas las peticiones almacenadas.

Los parámetros a valorar son los siguientes:

* Memoria consumida con todas las conexiones en espera.
* Threads abiertos.
* Tiempo que tarda el servidor en responder a todas las peticiones. IMPORTANTE: Este tiempo no es el tiempo total en responder al cliente, sino el tiempo en recorrer la lista de conexiones y devolver el control al hilo principal.

# Tecnologías probadas

## NodeJs

Node 0.10.6

Implementado con el http server de serie y un array de JS.

No se usa cluster puesto que el servidor solo tiene un core.

Ejecutado con todos los parámetros por defecto.

## Go

Go 1.1

Implementado con el http server de serie, channels y una lista.

Ejecutado con todos los parámetros por defecto.

## Java (Tomcat)

Oracle JDK1.7 + Tomcat 7 + Spring 3.2

Implementado con DeferredResult (Async Servlet 3.0) y un ArrayList

Parámetros de configuración:

* maxThreads=”1”
* maxConnections=”1000000”
* acceptorThreadCount=”1”
* acceptCount=”1000000”
* Xmx=5970

## Java (Jetty)

Oracle JDK1.7 + Jetty 8.1 + Spring 3.2

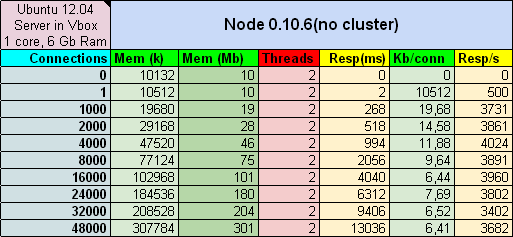
Implementado con DeferredResult (Async Servlet 3.0) y un ArrayList

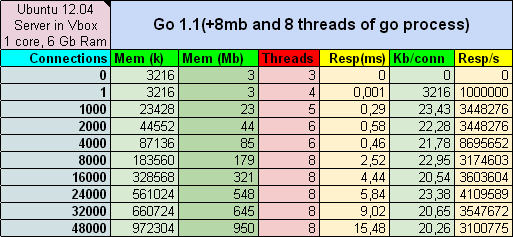
Parámetros de configuración:

* minThreads=10
* maxThreads=200
* Acceptors=2
* Xmx=4096

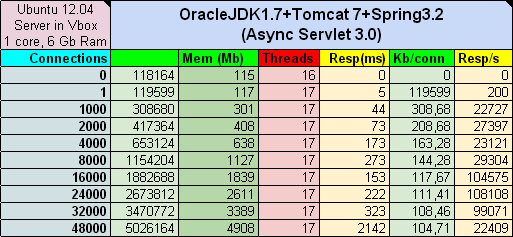
Nota: En la elaboración de este informe ya se había liberado la versión 9.0.3 de jetty. No obstante, se descartó su uso debido a un error conocido (Bug 408117) en la implementación de la especificación Servlet 3.0. En la versión 9.0.4 debería estar corregido.

# Resultados de la prueba





\*Additional 8mb and 8 threads consumed by “go” process



# 

# 

# 

# 

# 

# 

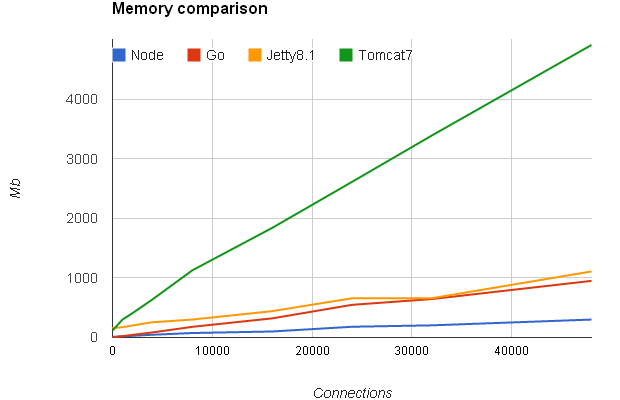
# 

# 

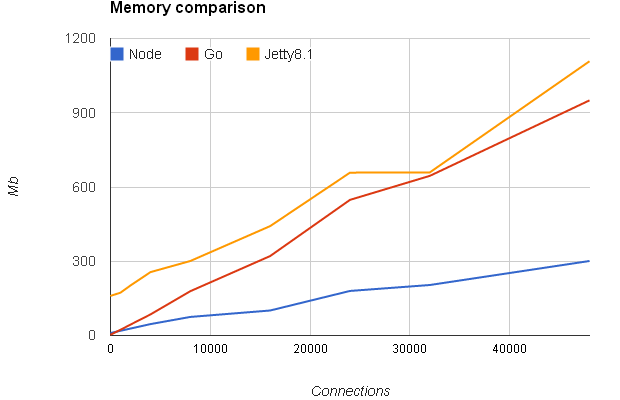
# 

# Comparativa

A continuación vamos a ver una gráfica comparativa con el uso de memoria de cada tecnología.



El consumo de memoria en Tomcat7 se dispara a los 5 Gb, vamos a quitarlo de la comparativa para ver mejor los otros datos.



En el gráfico anterior vemos claramente que NodeJs tiene el consumo de memoria más bajo con unos 300Mb y que Go y Jetty8.1 rondan 1Gb.

Ahora vamos a ver el tiempo que tardan en responder a todos los mensajes.

# 

* En este caso, el tiempo de respuesta de Go es sorprendentemente bajo. Node y Jetty son los más lentos y Tomcat da un resultado bastante bueno. IMPORTANTE: Estos tiempos no son el tiempo total en responder al cliente, sino el tiempo en recorrer la lista de conexiones y devolver el control al hilo principal.

# 

# 

# 

# 

# 

# 

# Conclusiones

A la vista de los resultados obtenidos, si necesitamos controlar el uso de memoria, NodeJs es el claro vencedor, con **6,41 Kb por conexión**. Por contra, el tiempo de respuesta es bastante “lento” con **3.682 respuestas por segundo**. No obstante, es un tiempo de respuesta más que suficiente para la mayoría de casos de uso.

Sin duda, el más rápido es Go con más de **3.000.000 respuestas por segundo** y un consumo de memoria moderado de **20,26 Kb por conexión.**

En el mundo Java, destaca el consumo desmesurado memoria de Tomcat 7 con **104,71 Kb por conexión** y una velocidad de respuesta de que oscila entre **20.000 y 100.000 respuestas por segundo.**

También en Java, el consumo de memoria de Jetty 8.1 es equiparable al de Go, con **23,63 Kb por conexión** y una velocidad de respuesta cercana a **3.000 respuestas por segundo**.

Se debe tener en consideración, que tanto NodeJs, como Go se han utilizado sin ningún framework adicional. Sin embargo las pruebas con Java utilizan el framework Spring 3.2, que proporciona todas las herramientas necesarias para el desarrollo web de manera eficiente.

IMPORTANTE: El tiempo de las respuestas no es el tiempo total en responder al cliente, sino el tiempo en recorrer la lista de conexiones y devolver el control al servidor. Además, hay que tener en cuenta que los clientes corrían en distintas máquinas virtuales pero en la misma máquina física. Esto implica que los retardos de latencia son prácticamente nulos, lo que facilita la comparación de la velocidad de los servidores, pero distorsiona el número de respuestas por segundo que se alcanzará en un escenario real.

# 

# 

# 

# 

# 

# 

# Trabajo futuro

Esta prueba está orientada a ofrecer un servicio de Long Pooling en un entorno Web. No obstante, estos resultados pueden ser extrapolados a cualquier tipo de servidor que ofrezca Long Pooling, por ejemplo para notificaciones push de aplicaciones móviles. Se podrían realizar pruebas específicas para diferentes entornos.

La prueba se ha limitado a usar un servidor con 1 core virtual y 6 Gb de Ram, llegando a un límite de 48.000 conexiones. Aumentando los recursos del servidor, se podría aumentar la prueba a un número mayor de conexiones. Teóricamente, se podrían atender 1.000.000 de conexiones simultáneas utilizando NodeJs con menos de 7 Gb de Ram.

Otra prueba deseable, sería la implementación de un servidor asíncrono pero con tráfico continuo en todas las conexiones. Por ejemplo, un servidor de descarga masiva. En este caso, la velocidad de Go podría ser un punto clave.

# Referencias

Código fuente: <https://github.com/bbva-innotech/testing>

<http://golang.org/>

<http://societyofcode.com/articles/benchmarks_nodejs_vs_go_vs_php_3_14_2013>

<https://sites.google.com/site/gopatterns/web/long-poll-server>

<http://techblog.safaribooksonline.com/2013/02/22/go-as-an-alternative-to-node-js-for-very-fast-servers/>

<http://blog.mixu.net/2011/02/02/essential-node-js-patterns-and-snippets/>

<http://zlayer.net/blog/?p=113>

<http://nodejs.org/>

<http://www.springsource.org/>

<http://blog.springsource.org/2012/05/06/spring-mvc-3-2-preview-introducing-servlet-3-async-support/>

<http://blog.springsource.org/2012/05/13/spring-mvc-3-2-preview-adding-long-polling-to-an-existing-web-application/>

<http://blog.springsource.org/2012/05/10/spring-mvc-3-2-preview-making-a-controller-method-asynchronous/>

<http://tomcat.apache.org/>

<http://www.eclipse.org/jetty/>