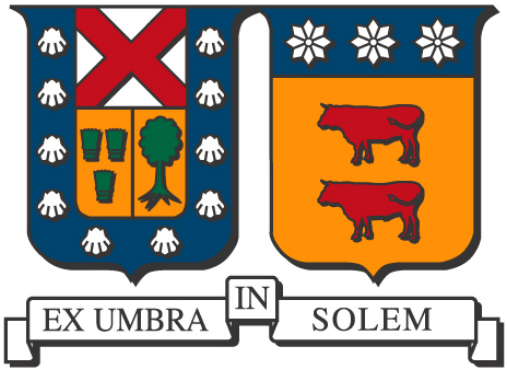
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE ELECTRÓNICA
VALPARAÍSO - CHILE



"ESTUDIO Y OPTIMIZACIÓN DE BACKEND PARA WEB SOCIAL GEORREFERENCIADA"

JORGE ANTONIO PERALTA TERÁN

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERO CIVIL TELEMÁTICO

PROFESOR GUÍA: TOMÁS ARREDONDO VIDAL.

PROFESOR CORREFERENTE 1: AGUSTÍN GONZÁLEZ V.

7 DE ENERO DE 2015

"The night is darkest just before the dawn".

Harvey Dent, The Dark Knight (2008)

Agradecimientos

Quisiera comenzar estas palabras, agradeciendo a mis padres, quienes a lo largo de estos años, han sido una ayuda fundamental para mis estudios, y en todas las dimensiones de mi vida. Quisiera agradecerles la buena disposición que siempre tienen, el tiempo que dedican para hacer de mi vida un poco más fácil, y el cariño que siempre me han entregado. Además, agradecerles los esfuerzos que han hecho durante su vida para darnos lo mejor posible. Debo decir que gracias a sus sacrificios, puedo ser la persona que escribe estas palabras.

También quiero agradecer a mi hermana, principalmente por la compañía, apoyo y alegrías que me ha entregado durante la vida. Espero que el destino nunca nos obligue a separarnos prematuramente.

Además quisiera agradecer a los profesores por las enseñanzas y consejos que me dieron. Y a los funcionarios de la UTFSM, sin ellos, la Universidad no sería lo mismo.

Finalmente, quisiera darles las gracias a mis amigos y compañeros, la vida es mejor si se esta bien acompañado.

Jorge Peralta Terán

Resumen

En el siguiente trabajo se buscó mejorar el rendimiento del servidor que mantiene el backend de la red social georreferenciada, Placetribe, en post de mantener la calidad de servicio para una mayor cantidad de usuarios.

Para ello, se utilizó una máquina virtual diseñada para ejecutar programas escritos en PHP *(HHVM)*, la cual para alcanzar un mayor rendimiento, utiliza un enfoque de compilación *JIT* (just-in-time). Y así poder mantener el código escrito para el backend de Placetribe.

Por otro lado, además, se utilizó un caché para almacenamiento de datos y objetos en memoria RAM *(memcached)*, y así reducir la cantidad de accesos a la base de datos.

Para comprobar la mejora del rendimiento del servidor, éste fue puesto a prueba utilizando el programa *Apache Benchmarking* (AB), y se compararon distintos escenarios: usando HHVM con memcached, solo HHVM, solo memcached y el escenario por default, sin las mejoras propuestas.

Esta memoria puede usarse como base, para proyectos de aplicaciones que deseen optimizar su backend escrito en PHP, o usando bases de datos. Además utilizarse como referencia para prever resultados y posibles problemas de implementación.

Abstract

The following work will seek to improve the performance of the backend server of Placetribe, a georeferenced social network, in order to maintain the quality of service for a greater number of users.

To do this, it's proposed the use of a virtual machine designed for executing programs written in PHP. This virtual machine uses a just-in-time compilation approach to achieve superior performance, while keeping the written code for the backend of Placetribe.

Moreover, is also proposed, a cache for storing data and objects in RAM memory, thereby reducing the number of accesses to the database.

To verify the improved performance of the server, this was tested using *Apache Benchmarking* (ab) and compared with the performance of the server, without the proposed improvements.

This work could be used to give an example for other projects, written in PHP or using databases, looking for improvements for their development. In addition it could be used to anticipate results and potential problems at the implementation.

Glosario

AB *Apache HTTP server benchmarking tool* - Herramienta utilizada en esta memoria para pruebas de benchmark.

Deployment Se refiere a las actividades necesarias para hacer que un sistema este disponible para su uso.

Bytecode También conocido como p-code, es una forma de set de instrucciones (más abstracto que código de máquina, diseñado para una ejecución eficiente de parte de un intérprete de software.

PaaS *Platform as a Service* - Plataforma como Servicio.

IaaS *Infrastructure as a Service* - Infraestructura como Servicio.

HHVM *HipHop Virtual Machine*.

IT *Information technology* - Tecnologías de la Información.

Índice general

1. Introducción **1**

1.1. Motivación y problema a resolver 1

1.2. Objetivos 2

1.2.1. Objetivos generales 2

1.2.2. Objetivos específicos 2

1.3. Estructura de la memoria 3

1.3.1. Estado del Arte 3

1.3.2. Discusión y propuesta de trabajo 3

1.3.3. FastCGI 3

1.3.4. Despliegue de la solución 3

1.3.5. Modificaciones hechas al código de Placetribe 4

1.3.6. Resultados 4

1.3.7. Conclusiones y trabajo a futuro 4

2. Estado del Arte **5**

2.1. Antecedentes 5

2.1.1. Compiladores PHP 5

2.1.2. Just-In-Time compilation 6

2.1.3. Caché de memoria 7

2.2. Herramientas disponibles 7

2.2.1. Phalanger 7

2.2.2. HHVM (HipHop Virtual Machine) 7

2.2.3. Memcached 8

3. Discusión y propuesta de trabajo **9**

4. FastCGI **13**

4.1. ¿Qué es FastCGI? . . . 13
4.1.1. ¿Cómo funciona? . . . 14

5. Despliegue de la solución **15**
5.1. Instalación HHVM . . . 15
5.2. Instalación Servidor web . . . 16
5.2.1. Instalación FastCGI . . . 17
5.2.2. Integración de FastCGI con Apache 2.2 . . . 18
5.3. Despliegue capa de negocios . . . 19
5.4. Iniciar el servidor . . . 20

6. Modificaciones hechas al código de Placetribe **22**

7. Resultados **26**
7.1. Consultas a medir . . . 27
7.1.1. Servicio: UserService . . . 27
7.1.2. Servicio: EventService . . . 28
7.1.3. Servicio: MarkerService . . . 30
7.1.4. Servicio: ChatService . . . 31
7.2. Gráficos de interés . . . 32
7.2.1. UserService::getUsersAsMemberByEventId . . . 33
7.2.2. EventService::getAllByUserId . . . 36
7.2.3. MarkerService::getByUserId . . . 39

8. Conclusiones y Trabajo a Futuro **42**
8.1. Análisis de objetivos . . . 42
8.2. Trabajo a Futuro . . . 43

Índice de figuras

3.1. Tecnologías utilizadas por Placetribe 9
3.2. Arquitectura de red de Placetribe 10
3.3. Diagrama de flujo del funcionamiento de Memcached 11

7.1. Comparación: Response times distribution 33
7.2. Comparación: Response times percentiles 33
7.3. Average response time: Escenario Default 34
7.4. Average response time: Escenario HHVM con Memcached 34
7.5. Response times vs Transactions per second: Escenario Default 35
7.6. Response times vs Transactions per second: Escenario HHVM con Memcached 35
7.7. Comparación: Response times distribution 36
7.8. Comparación: Response times percentiles 36
7.9. Average response time: Escenario Default 37
7.10. Average response time: Escenario HHVM con Memcached 37
7.11. Response times vs Transactions per second: Escenario Default 38
7.12. Response times vs Transactions per second: Escenario HHVM con Memcached 38
7.13. Comparación: Response times distribution 39
7.14. Comparación: Response times percentiles 39
7.15. Average response time: Escenario Default 40
7.16. Average response time: Escenario HHVM con Memcached 40
7.17. Response times vs Transactions per second: Escenario Default 41
7.18. Response times vs Transactions per second: Escenario HHVM con Memcached 41

Índice de tablas

7.1. *Resultados UserService::getById* 27
7.2. *Resultados UserService::getFriendsByUserIdAndCall* 28
7.3. *Resultados UserService::getUsersAsMemberByEventId* 28
7.4. *Resultados EventService::getById* 29
7.5. *Resultados EventService::getAllByUserId* 29
7.6. *Resultados EventService::getAllByUserIdAndMarkerId* 30
7.7. *Resultados MarkerService::getById* 30
7.8. *Resultados MarkerService::getByUserId* 31
7.9. *Resultados MarkerService::getByEventId* 31
7.10. *Resultados ChatService::getByUserId* 32

Capítulo 1

Introducción

Placetribe es una red social georreferenciada que tiene como objetivo entregar información sobre lugares y eventos. Permite a los usuarios crear lugares y eventos para compartirlos con sus amigos, compartir imágenes y chatear. El equipo de trabajo, inicialmente estaba compuesto por un profesor y 3 alumnos del departamento de Electrónica. Para luego ser conformado por un equipo multidisciplinario de 7 personas.

La aplicación hace uso de tecnologías como Apache, PHP y MySQL en el backend y Android e iOS en el Frontend. El código del backend está escrito en PHP y hace uso del modelo vista controlador (MVC). En este trabajo se buscó estudiar y entregar una propuesta para mejorar el rendimiento del servidor que mantiene la capa de negocios y la capa de datos de Placetribe, en post de ofrecer el servicio a una mayor cantidad de usuarios sin afectar la calidad de servicio.

1.1. Motivación y problema a resolver

En los sistemas actuales de arquitectura cliente-servidor, existe la tendencia a mantener la capa de negocios y la capa de datos en sistemas remotos a cargo de terceros, los cuales otorgan servicios como *PaaS* (Platform as a Service) o *IaaS* (Infrastructure as a Service) dependiendo de la cantidad de responsabilidades que se desean delegar, o de la necesidad de personalización de un sistema.

Uno de los motivos para hacer esto, es poder obtener capacidad de cómputo a menores precios sin tener que comprar los servidores y sin hacer el *deployment* correspondiente disminuyendo los gastos *CAPEX* (Capital expenditure). Otro de los motivos es el de delegar la administración y mantención de los sistemas y así disminuir los costos *OPEX* (Operating expense).

Además de los motivos financieros, tenemos los motivos que conciernen a la parte técnica de la empresa, como: *deployment* de ambientes de testing, desarrollo y producción en poco tiempo, disponibilidad, concentración de tiempo y esfuerzo del capital humano en el desarrollo de la aplicación.

Aunque existe un problema para este escenario. El poder de cómputo se transforma en un bien que debe ser usado de manera eficiente. Esto ya que muchas de las empresas que ofrecen los servicios *PaaS* e *IaaS* cobran en base al uso que el cliente le da [1] [2]. Por lo que en este trabajo se busca optimizar el backend de Placetribe, mejorando el rendimiento del servidor y la eficiencia en el uso de cómputo y acceso a memoria a través de tecnologías presentes en plataformas de Cloud Computing que permitan utilizar el código existente, sin hacer mayores intervenciones en éste.

1.2. Objetivos

1.2.1. Objetivos generales

Hacer un *deployment* de la plataforma backend de Placetribe, con tecnologías que mejoren el rendimiento del servidor.

1.2.2. Objetivos específicos

Los objetivos específicos del trabajo son los siguientes:

I. Generar un *deployment* de la plataforma backend de Placetribe, con las tecnologías previas al trabajo.

II. Mejorar el rendimiento del servidor, cambiando el interprete PHP de la plataforma, por un compilador JIT.

III. Reducir el tiempo de respuesta a través de un caché para consultas a la base de datos.

IV. Conocer el impacto generado por cada una de las tecnologías involucradas ejecutando pruebas de *benchmarking*, que comparen distintos escenarios.

1.3. Estructura de la memoria

Este trabajo está dividido en 7 capítulos: Estado del Arte, Discusión y propuesta de trabajo, *FastCGI*, Despliegue de la solución, Modificaciones hechas al código de Placetribe, Resultados y Conclusiones y trabajo a futuro.

1.3.1. Estado del Arte

Dados los antecedentes, conocimientos previos y la necesidad de tomar decisiones antes de comenzar el trabajo, se hace la investigación correspondiente de acuerdo a lo existente referente a este tema. Todo lo encontrado se desarrolla de forma resumida en este capítulo.

De esta forma, en un comienzo se explica el contexto en el que se encuentran los lenguajes de scripting como PHP versus lenguajes compilados, y se explican ventajas y desventajas, dejando en claro la necesidad de mejorar el rendimiento de los lenguajes interpretados (de scripting). Además se hace lo correspondiente a caché de memoria y explicando la necesidad de su uso.

1.3.2. Discusión y propuesta de trabajo

Luego de la investigación, se realiza una breve discusión para buscar las mejores herramientas y pasos a seguir en el desarrollo del trabajo. En este capítulo se expresan las decisiones tomadas y la propuesta de trabajo a seguir.

1.3.3. FastCGI

Es apropiado explicar un poco sobre FastCGI, ya que es un concepto importante en esta memoria, y para algunos, un poco difícil de comprender. Por lo que en este capítulo se explicará brevemente en que consiste.

1.3.4. Despliegue de la solución

En este capítulo se describen los aspectos necesarios para instalar y configurar las herramientas *HHVM* y *Memcached* en un ambiente *Linux CentOS 6.5*.

1.3.5. Modificaciones hechas al código de Placetribe

En este apartado, se detallan las modificaciones hechas al código de Placetribe, para utilizar Memcached, y se explica su uso.

1.3.6. Resultados

En este capítulo, se muestran los resultados obtenidos, a partir de las pruebas hechas con *AB* y revelan cuanto mejoró el desempeño del backend.

1.3.7. Conclusiones y trabajo a futuro

Finalmente, se sacan conclusiones respecto al trabajo previo, las expectativas y objetivos fijados, el desarrollo de las tecnologías y los resultados obtenidos. También se incluye un apartado de propuestas de trabajos próximos en los que se puede utilizar la presente memoria como base.

Capítulo 2

Estado del Arte

El estado del arte de esta memoria se divide en los conceptos, tecnologías y herramientas existentes que se pueden utilizar en el transcurso del trabajo y que se relacionan directamente con el problema a solucionar. Finalmente se decidirá que recursos se usarán para elaborar la solución.

2.1. Antecedentes

2.1.1. Compiladores PHP

Lenguajes de programación de scripts como Perl, Python, PHP, Ruby y Lua son ampliamente usados por los programadores de alto nivel ya que permiten una alta productividad. Esto se produce por distintos motivos. Por la amplia gama de librerías que disponen, reducen la cantidad de código a escribir, probar y depurar. Segundo la naturaleza dinámica de los lenguajes de scripting, provee flexibilidad y altos grados de polimorfismo dinámico. Por último y más importante, los lenguajes dinámicos son en general puramente interpretados, entregando facilidad para modificar y probar cambios en el código fuente. Puramente interpretado se refiere, a que no es necesaria una etapa de compilación para correr el programa luego de escribirlo o modificarlo. Esta es una ventaja relevante a la hora de desarrollar grandes sistemas, los cuales generalmente toman un tiempo considerable en estáticamente compilar y enlazar.

Por otro lado la mayor desventaja de estos lenguajes, en general, es el rendimiento. Esto es por sus características dinámicas y su implementación vía interpretación. Experimentos han comprobado que los programas tienden a usar un orden de magnitud más, en uso de CPU, cuando son implementados

usando lenguajes interpretados, comparado a cuando son implementados en lenguajes compilados [3].

El lenguaje PHP se implementó originalmente como un intérprete, y ésta es todavía la implementación más popular. Varios compiladores se han desarrollado para desacoplar el lenguaje PHP del intérprete. Las ventajas de compilación incluyen una mejor velocidad de ejecución, análisis estático, y la mejora de la interoperabilidad con el código escrito en otros idiomas [4].

Compiladores de PHP incluyen Phalanger, que compila PHP en Common Intermediate Language (CIL) código de bytes, y HipHop, desarrollada en Facebook y ahora disponible como código abierto, que transforma el script PHP a C++, y luego lo compila, lo que reduce el consumo de memoria del servidor hasta 5 veces y aumenta el web request throughput hasta en 9 veces [6].

El código fuente PHP, cuando se interpreta, se compila sobre la marcha a un formato interno que puede ser ejecutado por el motor de PHP [5]. Con el fin de acelerar el tiempo de ejecución y no tener que compilar el código fuente PHP cada vez que la página web se accede, los scripts PHP también se pueden implementar en un formato ejecutable usando un compilador de PHP.

2.1.2. Just-In-Time compilation

La compilación *Just-In-Time* es utilizada para acelerar la ejecución de bytecode comparado con la interpretación por si sola. El compilador JIT está a cargo de traducir la representación bytecode de la aplicación en código nativo, listo para ser ejecutado. Esta traducción ocurre justo en el momento (just-in-time) cuando una función está a punto de ser ejecutada cuando aún no existe código nativo todavía. Funciones que nunca se ejecutan, nunca son compiladas. Algunos sistemas compilan todas las funciones antes de tiempo *(AOT compilers)* con el costo de incrementar el tiempo de inicio. Otros sistemas tienen la política de primero interpretar el bytecode para evitar la penalización de compilar una función que solo se ejecuta pocas veces. En este caso, el compilador JIT correrá tan pronto como la máquina virtual haya detectado una *hot function*. Sistemas más avanzados tienen la política de que las funciones primero son interpretadas y luego cuando su frecuencia de llamados alcanza un umbral, son compiladas sin aplicar optimizaciones, para mantener el tiempo de compilación bajo; cuando alcanzan un segundo umbral, la función es compilada nuevamente, pero esta vez aplicando optimizaciones. El tiempo de overhead de compilación aumenta cada vez, pero esta la esperanza de recuperarlo cuando se corre una versión más rápida de la función varias veces [7].

2.1.3. Caché de memoria

Ha existido un gran incremento en el interés por sitios web que ofrecen Redes sociales y e-commerce durante los últimos años. Las redes sociales y los mercados financieros llevan el liderazgo generando gran cantidad de datos de manera dinámica. Típicamente estos datos son almacenados en bases de datos para una futura recuperación y análisis. Las bases de datos que usan lenguajes como SQL son bastante caras computacionalmente hablando. Los data-centers que mantienen sitios webs populares deben estar preparados para responder a millones de visitantes del sitio web. Y debido a la naturaleza de los datos, es inevitable que estos, sean accedidos. Es difícil mejorar el tiempo de respuesta para cada petición a la base de datos mientras se mantiene *ACID* (Atomicity, Consistency, Isolation and Durability) garantizado, especialmente cuando las lecturas son mezcladas con escrituras. Por lo cual se propone una nueva capa de memoria caché, memcached [8].

2.2. Herramientas disponibles

2.2.1. Phalanger

Phalanger es una implementación completa de PHP, reescrito en lenguaje C#. Se compone de compilador y runtime. Phalanger mejora la velocidad de ejecución, la seguridad y hace que la integración de PHP y .NET, muy simple.

Phalanger compila código fuente escrito en PHP al CIL Byte-code. Se encarga del inicio de un proceso de compilación que se completa con el componente compilador JIT de.NET Framework. No se ocupa de la generación de código nativo, ni de su optimización. Su finalidad es compilar los scripts PHP a .NET, las unidades lógicas que contiene el código CIL y meta-datos.

Por otro lado, trabaja como una extensión de ASP.NET que compila scripts automáticamente y permite usar PHP en servidores con ASP.NET habilitado [9].

2.2.2. HHVM (HipHop Virtual Machine)

HipHop es un motor de ejecución de código PHP. El proyecto comenzó en 2008 cuando ingenieros de Facebook comenzaron a trabajar en HPHP. El proyecto en un comienza era un compilador de PHP a C++ conocido como HPHPc y en su mejor momento, podía mostrar mejoras de hasta 6 veces comparado con Zend PHP [10]. Además junto con HPHPc se desarrollaron HPHPi (“modo desarrollador” de HipHop) y HipHop debugger conocido como HPHPd, que permitieron a los desarrolladores correr código PHP

a través de la misma lógica que HPHPc y al mismo tiempo depurar código PHP. Los desarrolladores podían usar relojes, breakpoints, etc. El rendimiento no era como el de HPHPc, pero de obtuvieron otros beneficios como mantener dos motores de ejecución al mismo tiempo para producción y desarrollo.

Todas estas herramientas en 2010 se transformaron a código abierto y fueron un éxito, permitiendo mejorar el rendimiento principalmente de Facebook y usando menos recursos. De todas maneras en 2013 estas herramientas quedaron obsoletas. Las razones fueron relacionadas principalmente a que las mejoras de rendimiento fueron cada vez menores, HPHPc no soportaba totalmente PHP y una serie de problemas de soporte entre HPHPc y HPHPd junto que HPHPi se desarrollaba lentamente. En comienzos de 2010 y para solucionar anticipadamente los posibles problemas de HPHPc, Facebook decidió crear una máquina virtual de PHP conocida como HHVM.

HHVM convierte código PHP a high-level bytecode, conocido comúnmente como lenguaje intermedio. Luego es convertido en código de máquina por un compilador JIT. En estos aspectos HHVM tiene similitudes con máquinas virtuales de otros lenguajes como C#/CLR y Java/JVM.

HHVM tiene casi soporte completo para la versión 5.4 de PHP. Como máquina virtual tiene la habilidad de usar información en vivo para producir código nativo más eficiente, dando a lugar a un mejor rendimiento del servidor web [11]. En el año 2012 HHVM alcanza igualdad en rendimiento con HPHPc. En 2013 la versión de producción de Facebook corre en HHVM, reemplazando a HPHPc.

2.2.3. Memcached

Memcached es un sistema distribuido de propósito general para caché basado en memoria, diseñado por Danga Interactive y que es muy usado en la actualidad por múltiples sitios web.

Memcached es empleado para el almacenamiento en caché de datos u objetos en la memoria RAM, reduciendo así las necesidades de acceso a un origen de datos externo (como una base de datos o una API). Memcached tiene versiones para Linux, Windows y MacOS y se distribuye bajo licencia de software BSD.

La extremadamente popular red social, Facebook, es un *heavy user* y contribuidor frecuente de memcached. Se ha reportado que en 2008, Facebook tiene 800 servidores Memcached con hasta 28 Terabytes de datos en su caché [12].

Su funcionamiento se basa en una tabla hash distribuida a lo largo de varios equipos. Conforme ésta se va llenando, los datos que más tiempo llevan sin ser utilizados se borran para dar espacio a los nuevos. Normalmente, las aplicaciones comprueban primero si pueden acceder a los datos a través de Memcached antes de recurrir a un almacén de datos más lento, como puede ser una base de datos.

Capítulo 3

Discusión y propuesta de trabajo

Considerando las tecnologías y herramientas disponibles presentadas, para este trabajo se desplegará un backend que imitará las condiciones del servidor de producción de Placetribe con las siguientes características:

1. Máquina Virtual (VirtualBox): 1 CPU 2.3Ghz emulado en un procesador i5-4310U, 2 Gb RAM
2. Sistema Operativo: CentOS 6.5
3. Servidor Web: Apache + PHP
4. Motor bases de datos: MySQL

Figura 3.1: *Tecnologías utilizadas por Placetribe*

Se propone mantener el sistema con esa configuración, para que este trabajo sea lo más fiel al servidor de producción.

Por la naturaleza de este trabajo, la arquitectura de red propuesta para este trabajo, es mantener la arquitectura de Placetribe, ya que los cambios y propuestas hechas, solo afectan directamente al backend, el cual posee la capa de negocios y datos en él. La **figura 7.18**, muestra la arquitectura de Placetribe (cliente-servidor) y grafica como trabaja el backend con la solución propuesta.

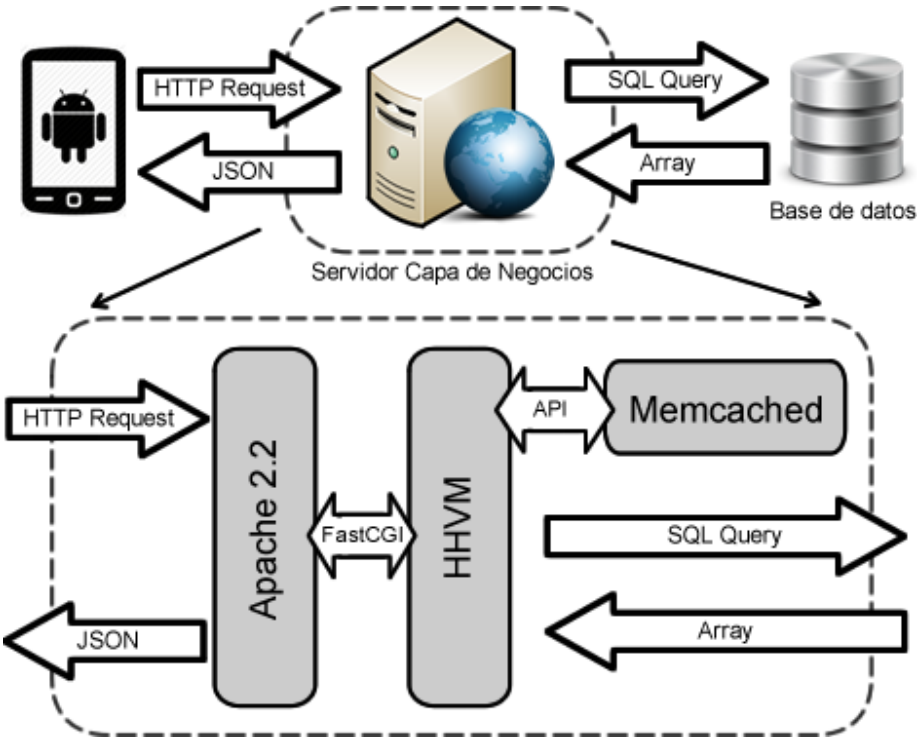


Figura 3.2: *Arquitectura de red de Placetribe*

La elección de HHVM, para usarse en este trabajo, radica principalmente en la promesa que hace, de mejorar cuantiosamente el desempeño del servidor. Esta promesa es de mejorar el desempeño del servidor casi 9 veces, para escenarios donde la demanda es bastante. Según lo documentado en el blog de HHVM [21], HHVM no tiene buen desempeño en escenarios con poca demanda hacia el servidor, si no que en los escenarios donde existe alta demanda, es donde se ven los beneficios de esta herramienta.

Entonces, el backend utilizará HHVM, como servidor para la interpretación/compilación de código PHP, el cual usará de interfaz al servidor web Apache, para comunicarse con el frontend de la aplicación, usando llamados HTTP. Y para la comunicación con la capa de datos, existe una capa extra entre HHVM y la base de datos, donde se ubica Memcached. HHVM todas hace consultas al caché, las cuales si no se encuentran allí se buscan en la base de datos y se almacenan en el caché para un futuro consumo. El

siguiente diagrama de flujo, muestra la lógica detrás del funcionamiento de Memcached.

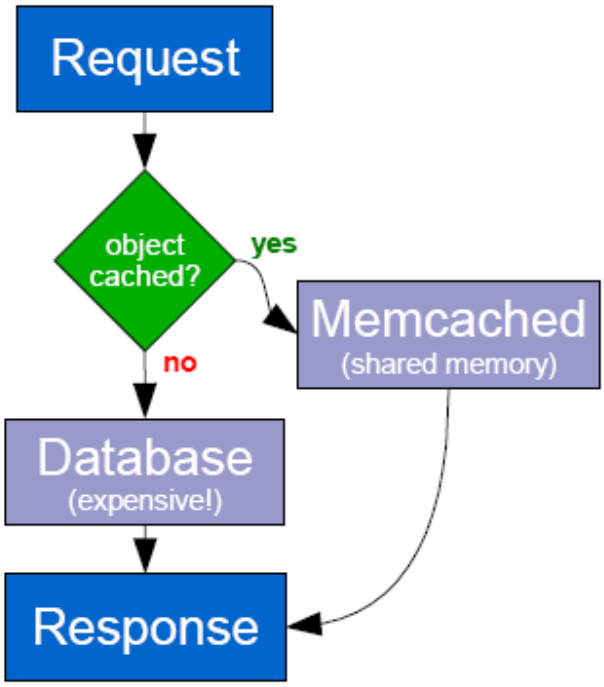


Figura 3.3: *Diagrama de flujo del funcionamiento de Memcached*
Fuente: http://cerberusweb.com

Una de las principales razones para la elección de estas herramientas es que pueden encontrarse en proveedores de Cloud Computing. Lo cual entrega a Placetribe la posibilidad de mover *workloads* a la nube en caso de necesitarlo en algún momento, y sin necesidad de hacer modificaciones mayores al trabajo ya realizado.

HHVM es una herramienta nueva pero que de igual manera puede encontrarse en proveedores de *PaaS*, como Heroku [13] [14] y CloudFoundry [15]. Memcached por su lado, se puede encontrar en proveedores como, Heroku [17], Amazon Web Services [18] y Pivotal [19], entre otros.

Que estas herramientas estén disponibles para usarse en Cloud Computing, no genera solo beneficios a nivel técnico, si no que principalmente a nivel económico. La adopción de tecnologías Cloud continua creciendo y acelerándose, y se debe principalmente a que las organizaciones grandes y pequeñas, intentan capitalizar usando los costos/beneficios de Cloud Computing. Parte de los beneficios que podemos encontrar, son [16]:

- Reducción de tiempo gastado en Administración IT.
- Requiere menos recursos IT internos.

- Mejora la disponibilidad del servicio.
- Produce mejoras en la seguridad de los procesos.
- Posibilidad de reinvertir en el negocio, a causa del ahorro producido por moverse a Cloud Computing

Uno podrá pensar de que estos beneficios son intrínsecos a Cloud Computing y que si no se hace uso de estas herramientas, de todas maneras tendrán los beneficios. Eso es correcto, pero cabe mencionar de que la gran mayoría, si no todos los proveedores de estos servicios cobran dependiendo del uso de computo que se utilice durante el tiempo de tarificación. Por lo que el uso eficiente de estos recursos vuelve a ser relevante. Muchos de estos proveedores poseen unidades de computo, por ejemplo Oracle posee OCPU, Heroku posee dynos, etc. El uso de las herramientas escogidas ayuda al ahorro en el consumo de estas unidades [20].

Capítulo 4

FastCGI

Es apropiado explicar un poco sobre FastCGI, ya que es un concepto importante en esta memoria, y para algunos, un poco difícil de comprender. Por lo que en este capítulo se explicará brevemente en que consiste.

4.1. ¿Qué es FastCGI?

FastCGI es un protocolo binario para la comunicación entre una aplicación servidor (que por ejemplo: interprete código PHP), y un servidor web (Apache, Nginx Lighttpd, etc). Con el soporte de FastCGI, se puede usar HHVM detrás de cualquier servidor web, que también lo soporte. Con esto, se distribuyen las tareas, y permite que el servidor web se haga cargo de manejar los requerimientos del protocolo HTTP. Esto permite que el desarrollo de HHVM esté enfocado exclusivamente en su labor principal, ejecutar código PHP.

FastCGI es una variación de *Common Gateway Interface* (CGI), pero apunta principalmente a reducir el overhead asociado con la comunicación entre el servidor web y los programas CGI. Para comunicar las aplicaciones CGI con el servidor web, por cada una de las aplicaciones, se crea un proceso distinto durante cada petición que se hace, lo que facilita su implementación pero limita su eficiencia y escalabilidad. Con cargas considerables, el overhead creado por el sistema operativo por la creación y destrucción de proceso, se vuelve significativo.

4.1.1. ¿Cómo funciona?

En lugar de crear un proceso por cada peticón, FastCGI usa procesos persistentes para manejar una serie de peticiones. Para procesar una petición, el servidor web envía información del sistema, y la petición request misma a un proceso FastCGI a través de un socket o a través de una conexión TCP. La conexión puede cerrarse después de una petición, pero los procesos del servidor web y del servicio FastCGI, persisten.

Cada proceso FastCGI, puede manejar muchas peticiones durante el tiempo, por lo que evita el overhead generado por la creación y destrucción de cada proceso por petición. El procesamiento de múltiples peticiones simultánes puede ser logrado de varias maneras: usando una sola conexión con multiplexación interna (manejo de varias peticiones con una sola conexión); usando múltiples conexiones; o una combinación de estas técnicas, que permitiría la configuración de múltiples servidores, aumentando la estabilidad y escalabilidad.

Administradores de servidores web y programadores, pueden encontrar múltiples beneficios de la separación del servidor web de sus aplicaciones, versus el uso de interpretes embebidos. Esta separación permite al servidor y aplicaciones, ser reiniciados independientemente (algo considerable para servidores de bastante utilización). Permite además la implementación de políticas de seguridad por aplicación, por servidor. Otro beneficio es la diferenciación de distintos tipos de consultas y su distribución a diversos servidores FastCGI optimizados para manejar esas consultas específicas.

Capítulo 5

Despliegue de la solución

En este capítulo se describen los aspectos necesarios para instalar y configurar las herramientas *HHVM* y *Memcached* en un ambiente *Linux CentOS 6.5*. El motivo para esto, es entregar una documentación completa del proceso, ya que la documentación existente no se encuentra centralizada, por lo que muchos pasos se deben investigar por cuenta propia, creando un proceso bastante engorroso en el que un usuario básico podría perder bastante tiempo, intentando lograr que funcione en su sistema.

Se asume que el lector tiene conocimientos básicos de Linux, por lo que los pasos iniciales de configuración de red, configuración de SSH, e instalación de herramientas básicas como editores se omitirán.

5.1. Instalación HHVM

Para la instalación de HHVM en CentOS 6.5, basta con seguir los pasos indicados por la documentación oficial [22], que se detallan a continuación:

Luego de las configuraciones básicas para tener un sistema con CentOS operando, se actualiza el sistema usando:

```
yum update
```

Se instala el archivo de configuración para hacer uso del repositorio *epel*:

```
sudo rpm -Uvh http://dl.fedoraproject.org/pub/epel/6/x86_64/
epel-release-6-8.noarch.rpm
```

Luego se instalan los paquetes que HHVM necesita:

```
sudo yum install libmcrypt-devel glog-devel jemalloc-devel \
tbb-devel libdwarf-devel mysql-devel libxml2-devel libicu-devel \
pcre-devel gd-devel boost-devel sqlite-devel pam-devel bzip2-devel \
oniguruma-devel openldap-devel readline-devel libc-client-devel \
libcap-devel libevent-devel libcurl-devel libmemcached-devel
```

Ahora, se instala el archivo de configuración para usar el repositorio *gleez*:

```
sudo rpm -Uvh http://yum.gleez.com/6/x86_64/gleez-repo-6-0.el6.
noarch.rpm
```

Si por algún motivo se obtiene un mensaje con error 403, que diga: *you have been banned from the site*, entonces se requiere descargar el archivo desde un navegador y transferirlo al servidor, luego de esto, ejecutar:

```
sudo rpm -Uvh gleez-repo-6-0.el6.noarch.rpm
```

Después, instalar el archivo de configuración del repositorio *remi*:

```
sudo rpm -Uvh http://rpms.famillecollet.com/enterprise/6/remi/x86_
64/remi-release-6.5-1.el6.remi.noarch.rpm
```

Se instalan los últimas dependencias:

```
yum --enablerepo=remi install libwebp mysql mysql-devel mysql-lib
```

Y finalmente, se instala HHVM:

```
yum --nogpgcheck install hhvm
```

El flag *–nogpgcheck* se utiliza ya que algunos de los paquetes instalados por HHVM no están firmados.

Luego de estos pasos, se obtiene una instalación de HHVM, pero aún se necesitan otros componentes.

5.2. Instalación Servidor web

La instalación de Apache es bastante sencilla, basta con ejecutar el comando:

```
yum install -y httpd
```

Ahora, se debe habilitar el servicio, usando:

```
chkconfig httpd on
```

En el archivo de configuración de Apache:

```
/etc/httpd/conf/httpd.conf
```

Para evitar enviar información sensible, que puede ayudar a terceros a usar exploits, se recomienda editar la variable *ServerTokens* [23], quedando:

```
ServerTokens Prod
```

Además, para el código de Placetribe, es importante que el virtualhost que le de servicio, habilite la directiva *AllowOverride*. Solo se puede usar en secciones *Directory*. Esto permite al servidor encontrar los archivos .htaccess y ejecutar sus instrucciones [24].

Luego de ubicar los archivos en la carpeta donde se alojará, es importante que se cambie el contexto de los archivos, para que Apache pueda utilizarlos. Para ello, estando en la ubicación de los archivos, se ejecuta:

```
chcon -Rv --type=httpd_sys_content_t *
```

Finalmente, habilitar el puerto 80 en el firewall. En este caso, *iptables*.

```
iptables -I INPUT 5 -i eth0 -p tcp --dport 80 -m state\
--state NEW,ESTABLISHED -j ACCEPT
```

Y guardar permanentemente la nueva regla:

```
service iptables save
```

Para CentOS 6.5 no existe posibilidad de instalar Apache 2.4 (disponible en CentOS 7), por lo que se instala la versión correspondiente a este sistema operativo, Apache 2.2, pero esta versión no tiene integración incluida con *FastCGI*, que es el protocolo que utilizaremos para comunicar Apache con HHVM, por lo que debemos instalar esta característica.

5.2.1. Instalación FastCGI

Para instalar *FastCGI* y poder utilizarlo junto con apache algunos paquetes son requeridos. Para instalarlos, ejecutar:

```
yum install libtool httpd-devel apr-devel apr
```

Luego, descargar la última versión del código fuente de FastCGI:

```
cd /opt
wget http://www.fastcgi.com/dist/mod_fastcgi-current.tar.gz
```

Descomprimir el tar ball:

```
tar -zxvf mod_fastcgi-current.tar.gz
cd mod_fastcgi-2.4.6/
```

Copiar el archivo Makefile.AP2, para usarlo como *makefile*:

```
cp Makefile.AP2 Makefile
```

Editar el archivo *Makefile*, en la linea donde se declara la variable *top_dir*, quedando:

- Para sistemas de 32 bit, usar:

```
top_dir=/usr/lib/httpd
```

- Para sistemas de 64 bit, usar:

```
top_dir=/usr/lib64/httpd
```

Luego compilar e instalar

```
make install
```

Abrir el archivo de configuración de FastCGI:

```
vim /etc/httpd/conf.d/mod_fastcgi.conf
```

Y agregar la siguiente linea:

```
LoadModule fastcgi_module modules/mod_fastcgi.so
```

Guardar y cerrar. Iniciar el servicio:

```
service httpd start
```

5.2.2. Integración de FastCGI con Apache 2.2

Esta sección se refiere a los pasos que se deben realizar para poder hacer uso de FastCGI, en este caso para comunicar HHVM con Apache.

Como lectura adicional, se le recomienda al lector de esta memoria revisar:
`http://whocares.de/fastcgiexternalserver-demystified/all/1/`

En primer lugar, HHVM pide habilitar los módulos *alias* y *action*, y además usaremos el módulo de virtualhosts, los cuales están habilitados en la configuración de apache por defecto en este caso. En caso de no tener habilitadas estos módulos, editar el archivo de configuración usando:

```
vim /etc/httpd/conf/httpd.conf
```

En las lineas: 182, 185 y 188 respectivamente, habilitar los módulos que usaremos, quedando:

```
LoadModule vhost_alias_module modules/mod_vhost_alias.so
LoadModule actions_module modules/mod_actions.so
LoadModule alias_module modules/mod_alias.so
```

Además en nuestro caso, no se instaló PHP, por lo que se requiere cambiar la directiva *DirectoryIndex*.

```
DirectoryIndex index.html index.html.var index.php index.hh
```

Agregar a la configuración del virtualhost las siguientes lineas.

```
<IfModule mod_fastcgi.c>
    <FilesMatch \.php$>
        SetHandler hhvm-php-extension
    </FilesMatch>

    <FilesMatch \.hh$>
        SetHandler hhvm-hack-extension
    </FilesMatch>

    Alias /hhvm /hhvm
    Action hhvm-php-extension /hhvm virtual
    Action hhvm-hack-extension /hhvm virtual

    FastCgiExternalServer /hhvm -host 127.0.0.1:9000
-pass-header Authorization -idle-timeout 300
</IfModule>
```

Esto hará que todos los archivos *.php y *.hh vayan a través del servidor FastCGI (HHVM). Como acotación, **no funcionará** si es que */hhvm* es un directorio real y se están usando reglas *mod_rewrite* [25].

5.3. Despliegue capa de negocios

Como se mencionó en el **Capítulo 3**, el motor utilizado será MySQL, lo cual para los efectos de este trabajo, no tiene mayor impacto, ya que las consultas a la base de datos, solo se efectúan cuando la

consulta no esta en caché. Esto ocurre cuando la consulta se hace por primera vez, o cuando expira la consulta en caché.

Esta sección explicara brevemente el despliegue de la capa de negocios. En el **Capítulo 6**, se explicarán las modificaciones hechas al código del backend para poder funcionar con las herramientas propuestas.

Ya que previamente se instaló el cliente MySQL, solo falta instalar el servidor, para ello, basta ejecutar:

```
yum --enablerepo=remi install mysql-server
```

Es necesario instalar desde el repositorio *remi*, ya que es el repositorio que mantiene las dependencias con los programas instalados para usar HHVM. Luego se inicia el servicio, usando:

```
service mysqld start
```

Se establece la contraseña del usuario root del motor:

```
/usr/bin/mysqladmin -u root password t00r
```

Se habilita el servicio para que inicie al prenderse el servidor, usando:

```
chkconfig mysqld on
```

Previamente se hicieron dumps, de la base de datos de Placetribe, por lo que luego de transferir el archivo, se realiza la importación de este al motor:

- Primero se crea la base de datos placetribe:

```
mysql -u root -pt00r
   mysql> CREATE DATABASE placetribe;
   mysql> exit
```

- Finalmente, se importa la base de datos:

```
mysql -u root -p placetribe < bd_placetribe-2014100902_latin1.sql
```

Para implementar Memcached, en este caso, basta solo instalar su paquete. HHVM ya tiene las librerías necesarias para interactuar con Memcached.

```
yum install memcached
```

5.4. Iniciar el servidor

Cabe mencionar, que hasta el momento, todos los servicios están iniciados (excepto por hhvm), por lo que luego de tener todo configurado, algunos de estos se deben volver a iniciar. Es importante mencionar

que existe un orden para realizar esto, el cual permite evadir algunos errores que entrega el sistema. El orden para realizar esto sería:

1. Verificar que mysqld, este corriendo.
2. Verificar que memcached, este corriendo.
3. Iniciar servidor HHVM

   ```
   hhvm --mode server -vServer.Type=fastcgi -vServer.Port=9000
   ```

4. Reiniciar httpd

HHVM puede iniciar en modo *daemon*, para lo cual se debe reemplazar el valor en el flag *–mode*.

```
hhvm --mode daemon -vServer.Type=fastcgi -vServer.Port=9000
```

En caso de usarse *mod_rewrite* (que es nuestro caso) o recibir un error 404, agregar:

```
-vServer.FixPathInfo=true
```

Pero además puede iniciarse usando un UNIX socket:

```
hhvm --mode server -vServer.Type=fastcgi -vServer.FileSocket=/var/run
/hhvm/socket
```

En caso de usarse este modo, se debe reemplazar en el archivo de configuración de Apache, en el virtualhost correspondiente, la linea en que se declara *FastCGIExternalServer*, por:

```
FastCGIExternalServer /hhvm -socket /var/run/hhvm/socket -pass-header
Authorization -idle-timeout 300
```

Con esto, se obtiene un ambiente funcional, en el que se puede hacer pruebas. Pero antes, en el siguiente capítulo, se explicarán las modificaciones hechas al código para lograr su funcionamiento.

Capítulo 6

Modificaciones hechas al código de Placetribe

En este capítulo, se explicarán los cambios necesarios para hacer que el código de Placetribe funcione con Memcached. La idea de estas modificaciones, es que sean menores, para mantener la mayor cantidad de código posible y facilitar su implementación.

Hay que considerar de que Placetribe esta escrito usando programación orientada a objetos, lo cual resulta muy conveniente a la hora de realizar modificaciones, si es que el código fue escrito ordenadamente, el cual es el caso.

El código que se agrega, básicamente, son dos archivos, *MemcachedConfig.php* y *DataBaseManager.php*

En *MemcachedConfig.php* se utilizó una clase [26] básica que interactúa con la clase *Memcache* [27], para mantener algunas variables estáticas, para las pruebas de benchmark. Tiene los mismos métodos básicos, pero además realiza las comprobaciones necesarias, en caso de no tener las librerías de *Memcache*, instaladas.

```
1  <?php
2  class CacheMemcache {
3      var $bEnabled = false; // Memcache habilitado?
4      var $oCache = null;
5
6      // constructor
7      function CacheMemcache() {
```

```
8           if (class_exists('Memcache')) {
9               $this->oCache = new Memcache();
10              $this->bEnabled = true;
11              if (! $this->oCache->pconnect('localhost', 11211))  { // En
                    lugar de 'localhost' puede ir una IP
12                  $this->oCache = null;
13                  $this->bEnabled = false;
14              }
15          }
16      }
17
18      // get data from cache server
19      function getData($sKey) {
20          $vData = $this->oCache->get($sKey);
21          return false === $vData ? null : $vData;
22      }
23
24      // save data to cache server
25      function setData($sKey, $vData, $TimeToLive=120) { // TTL medido en
            segundos
26          //Usar MEMCACHE_COMPRESSED, en vez de '0', para almacenar el item
                compreso (usa zlib).
27          return $this->oCache->set($sKey, $vData, 0, $TimeToLive);
28      }
29
30      // delete data from cache server
31      function delData($sKey) {
32          return $this->oCache->delete($sKey);
33      }
34  }
35  ?>
```

Para el caso de *DataBaseManager.php*, donde se hicieron las reales modificaciones al código de Placetribe, se decidió utilizar un enfoque, que permita al código funcionar con Memcached, y sin Memcached. Las intervenciones se hicieron en los métodos *execute* y *silentExecute* y siguen con el concepto explicado en el **Capítulo 3**, en el que se primero se verifica si la query se encuentra en el caché, y luego si es que se encuentra, se retorna el arreglo correspondiente. Si no la query no se encuentra en el caché, se hace el

llamado a la base de datos, después se guarda el arreglo en caché, para una futura consulta.

A continuación solo se muestra la consulta *execute*, ya que *silentExecute* es básicamente la misma consulta, pero sin arrojar *EmptyResponseException* cuando no hay filas que mostrar.

```
1  public function execute($query, $useMemcached = FALSE, $ttlMemcached = 300)
       {
2        if($useMemcached){
3            $cache_array = array();
4            $keyCache = md5($query);
5            // Verificar si es que la query esta en RAM.
6            $cache = new CacheMemcache();
7            if ($cache->bEnabled){ // Si es que Memcache esta instalado y
                 conectado; Opero.
8                $cache_array = $cache->getData($keyCache);
9                if($cache_array != null){
10                   return $cache_array;
11               }
12           }
13       }
14       $return_array = array();
15       $result = mysql_query($query);
16
17       if (!$result) {
18           throw InvalidQueryException::getInstance($query, mysql_errno(),
                 mysql_error());
19       }
20       while ($row = @mysql_fetch_object($result)) {
21           $return_array[] = $row;
22       }
23       if (count($return_array) == 0) {
24           throw EmptyResponseException::getInstance($query);
25       }
26       if ($useMemcached && $cache->bEnabled){
27           $cache->setData($keyCache, $return_array, $ttlMemcached);
28       }
29       return $return_array;
30   }
```

Las modificaciones hechas, fueron pensadas para que al alterar el código de Placetribe, fuera lo menos posible, a continuación se detalla lo simple que es usar Memcached con las modificaciones hechas. Cabe mencionar de que el código, otorga la elección al desarrollador decidir que consulta irá al caché.

Para dejar en caché las consultas deseadas, basta con modificar el DAO correspondiente, y cuando se haga la invocación del método *execute* o *silentExecute* se agreguen 2 argumentos al final: TRUE, parar declarar que se desea dejar la consulta en caché, y su TTL. En la línea 4 del siguiente snippet, se ejemplifica lo explicado anteriormente.

```
1       public function getById($user_id) {
2           $dbm = DataBaseManager::getInstance();
3           $user_id = $dbm -> escapeVar($user_id);
4           $result = $dbm -> execute('SELECT * from user_t WHERE user_id ='.
                $user_id,TRUE, 120);
5           $data = $result[0];
6           $data = get_object_vars($data);
7           $session_id = $data['session_id'];
8
9           $user = User::getInstance() -> setData($data);
10
11          //Check if it's online or offline
12          if ($session_id == NULL)//OFFLINE
13              $user -> setMarkerStatus(User::$USER_STATUS_INACTIVE);
14
15          return $user;
16           }
```

Capítulo 7

Resultados

En el presente capítulo, se exponen los resultados de las pruebas realizadas a los servidores utilizados durante el trabajo. Las pruebas que se han definido según los objetivos detallados en **Capítulo 1**: *Introducción*.

Placetribe, al ser una red social, tiene usuarios, que generan nuevo contenido, usando las distintas entidades que se encuentran en Placetribe, pero más que generar contenido, lo consultan. La filosofía detrás de Placetribe, es ser una plataforma donde las personas puedan ver información de lugares donde reunirse, donde puedan ver contenido sobre eventos y lo que está ocurriendo en ellos. El contenido es creado por los usuarios, pero también es generado por empresas interesadas en tener exposición frente a un público joven (público objetivo de Placetribe).

Es por esto que se ha decidido que las pruebas de benchmark, serán generadas pensando en el público que consumirá el contenido. La razón de esto, es principalmente la misma de este trabajo de memoria, mejorar el desempeño del servidor, pensando en el futuro de Placetribe, donde existirá una mayor cantidad de usuarios, que consumirá el contenido desplegado por esta plataforma y se debe dar la misma calidad de servicio para todos ellos.

Para comprobar qué tan efectivos son los ajustes realizados al backend de Placetribe, se han configurado cuatro escenarios, a los que se han hecho pruebas:

1. Sistema con configuraciones por default
2. Sistema solo con Memcached como optimización
3. Sistema solo con HHVM como optimización
4. Sistema con HHVM y Memcached.

La prueba a realizar consiste en 30 iteraciones de llamadas por *AB* al servidor. Esto para tener las suficientes muestras y poder utilizar el Teorema Central de Límite, que indica que para una muestra grande, implica que tiene una distribución aproximadamente Normal, sin importar la naturalez de la distribución poblacional [28]

```
ab -c 100 -n 1000 -q -g resultados/tmp.tsv http://rest.placetribe.
local/<path_de_la_consulta>
```

La elección de los flags de concurrencia y de número de peticiones, se fijó mediante experimentación, ya que el servidor de prueba al no tener las mismas características de procesamiento que el servidor de producción, no responderá de igual manera a la carga que se le enfrente.

7.1. Consultas a medir

Se eligieron 10 consultas, las cuales se prevé que tendrán mayor uso y además son las más importantes para el negocio de Placetribe. A continuación se detallan los resultados para cada una de ellas en los distintos escenarios.

7.1.1. Servicio: UserService

7.1.1.1. getById

- URL REST: `http://rest.placetribe.local/users/17`
- Variable requerida: user_id
- Valor entregado a la variable: 17

Tiempo de respuesta por petición [milisegundos]			
Escenario	**Media Aritmética**	**Desviación Estandar**	**Intervalo de confianza (%95)**
Default	2464,7239	344,4059176	3,89724823
Solo Memcached	2478,3046	371,8480232	4,207779183
Solo HHVM	382,3057667	54,66786344	0,618613744
HHVM y Memcached	404,7586	56,63422146	0,640864771

Tabla 7.1: *Resultados UserService::getById*

7.1.1.2. getFriendsByUserIdAndCall

- URL REST: `http://rest.placetribe.local/users/17/friends/all`
- Variable requerida: user_id; call
- Valor entregado a la variable: 17; all

Tiempo de respuesta por petición [milisegundos]			
Escenario	**Media Aritmética**	**Desviación Estandar**	**Intervalo de confianza (%95)**
Default	2643,1889	332,5960011	3,763608901
Solo Memcached	2677,374967	386,6555632	4,375339193
Solo HHVM	451,0254333	67,7553064	0,766709381
HHVM y Memcached	537,6844	81,97423369	0,927608734

Tabla 7.2: *Resultados UserService::getFriendsByUserIdAndCall*

7.1.1.3. getUsersAsMemberByEventId

- URL REST: `http://rest.placetribe.local/users/all/events/50000`
- Variable requerida: event_id
- Valor entregado a la variable: 50000

Tiempo de respuesta por petición [milisegundos]			
Escenario	**Media Aritmética**	**Desviación Estandar**	**Intervalo de confianza (%95)**
Default	2594,126333	342,5918305	3,87672028
Solo Memcached	2546,6267	421,5504389	4,770204628
Solo HHVM	407,6990333	65,20088259	0,737803886
HHVM y Memcached	429,2426	66,85957836	0,756573451

Tabla 7.3: *Resultados UserService::getUsersAsMemberByEventId*

7.1.2. Servicio: EventService

7.1.2.1. getById

- URL REST: `http://rest.placetribe.local/events/50000`

- Variable requerida: event_id
- Valor entregado a la variable: 50000

Tiempo de respuesta por petición [milisegundos]			
Escenario	**Media Aritmética**	**Desviación Estandar**	**Intervalo de confianza (%95)**
Default	2622,9046	320,7744379	3,629837778
Solo Memcached	2582,105933	400,6365355	4,533545881
Solo HHVM	409,6272667	70,13798577	0,793671441
HHVM y Memcached	494,5642	83,04193485	0,939690688

Tabla 7.4: *Resultados EventService::getById*

7.1.2.2. getAllByUserId

- URL REST: `http://rest.placetribe.local/events/user/17`
- Variable requerida: user_id
- Valor entregado a la variable: 17

Tiempo de respuesta por petición [milisegundos]			
Escenario	**Media Aritmética**	**Desviación Estandar**	**Intervalo de confianza (%95)**
Default	2639,927833	321,6011539	3,639192778
Solo Memcached	2599,0076	395,7613631	10,96974373
Solo HHVM	455,0346667	91,89895962	1,039915517
HHVM y Memcached	561,8193333	249,3000736	2,821044068

Tabla 7.5: *Resultados EventService::getAllByUserId*

7.1.2.3. getAllByUserIdAndMarkerId

- URL REST: `http://rest.placetribe.local/events/user/2/marker/0`
- Variable requerida: user_id; marker_id
- Valor entregado a la variable: 2; 0

Tiempo de respuesta por petición [milisegundos]			
Escenario	**Media Aritmética**	**Desviación Estandar**	**Intervalo de confianza (%95)**
Default	3095,585567	342,7880612	3,878940799
Solo Memcached	3044,9654	516,514607	14,31679138
Solo HHVM	822,4841333	625,6848549	7,080160558
HHVM y Memcached	908,6012667	599,5544898	6,784473074

Tabla 7.6: *Resultados EventService::getAllByUserIdAndMarkerId*

7.1.3. Servicio: MarkerService

7.1.3.1. getById

- URL REST: `http://rest.placetribe.local/markers/14`
- Variable requerida: marker_id
- Valor entregado a la variable: 14

Tiempo de respuesta por petición [milisegundos]			
Escenario	**Media Aritmética**	**Desviación Estandar**	**Intervalo de confianza (%95)**
Default	2511,049767	423,2643388	4,789598875
Solo Memcached	2570,4166	360,2001964	9,984056594
Solo HHVM	742,0005333	782,1532596	8,850734704
HHVM y Memcached	847,6710667	917,2444681	10,37940755

Tabla 7.7: *Resultados MarkerService::getById*

7.1.3.2. getByUserId

- URL REST: `http://rest.placetribe.local/markers/user/15`
- Variable requerida: user_id
- Valor entregado a la variable: 15

Tiempo de respuesta por petición [milisegundos]			
Escenario	**Media Aritmética**	**Desviación Estandar**	**Intervalo de confianza (%95)**
Default	3291,2633	481,6729096	5,450541929
Solo Memcached	3914,9132	555,1124211	15,38664855
Solo HHVM	456,9243667	81,79789653	0,925613328
HHVM y Memcached	1768,880067	298,111344	3,37338544

Tabla 7.8: *Resultados MarkerService::getByUserId*

7.1.3.3. getByEventId

- URL REST: `http://rest.placetribe.local/markers/event/50268`
- Variable requerida: event_id
- Valor entregado a la variable: 50268

Tiempo de respuesta por petición [milisegundos]			
Escenario	**Media Aritmética**	**Desviación Estandar**	**Intervalo de confianza (%95)**
Default	2470,6846	347,0858046	3,92757345
Solo Memcached	2503,48	419,9238803	11,63948223
Solo HHVM	379,2944	51,88144626	1,438053896
HHVM y Memcached	398,6846	54,51249623	1,51098154

Tabla 7.9: *Resultados MarkerService::getByEventId*

7.1.4. Servicio: ChatService

7.1.4.1. getByUserId

- URL REST: `http://rest.placetribe.local/chat/user/15`
- Variable requerida: user_id
- Valor entregado a la variable: 15

Tiempo de respuesta por petición [milisegundos]			
Escenario	**Media Aritmética**	**Desviación Estandar**	**Intervalo de confianza (%95)**
Default	2970,0625	475,7005071	5,382959075
Solo Memcached	2572,4096	387,907395	10,75204684
Solo HHVM	186,2486333	104,0948981	1,177923017
HHVM y Memcached	453,5846	64,81559268	1,796563556

Tabla 7.10: *Resultados ChatService::getByUserId*

7.2. Gráficos de interés

En la siguiente sección se muestran gráficos que reflejan de otro modo las mejoras y los resultados obtenidos de las pruebas de benchmarking. Se utilizaron los resultados de las 3 funciones más importantes para la plataforma de Placetribe. Para obtener los gráficos, se utilizó `http://loadosophia.org/` y el output de *AB*, en formato *tsv*.

- UserService::getUsersAsMemberByEventId
- EventService::getAllByUserId
- MarkerService::getByUserId

7.2.1. UserService::getUsersAsMemberByEventId

7.2.1.1. Response times distribution

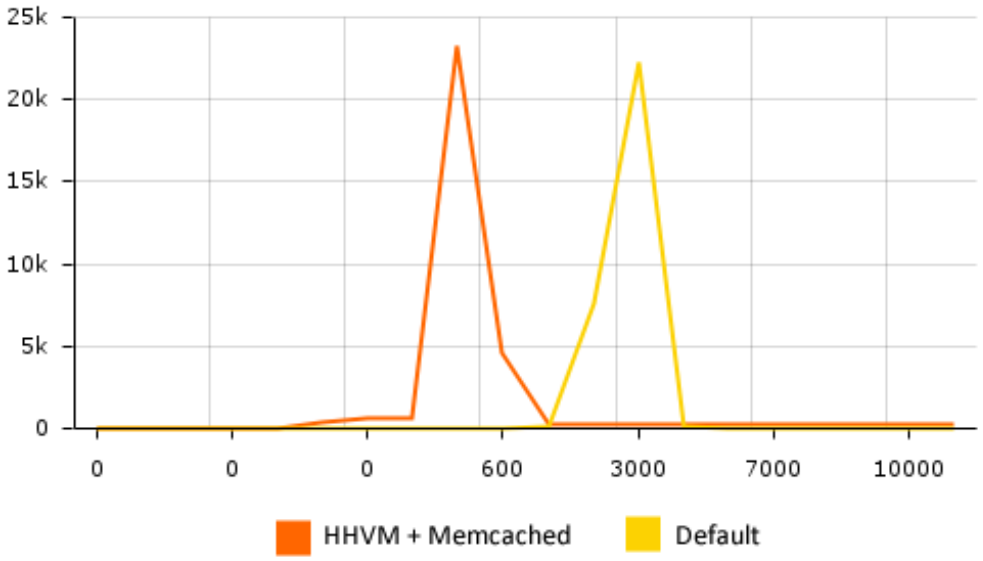


Figura 7.1: *Comparación entre ambos escenarios*

7.2.1.2. Response times percentiles

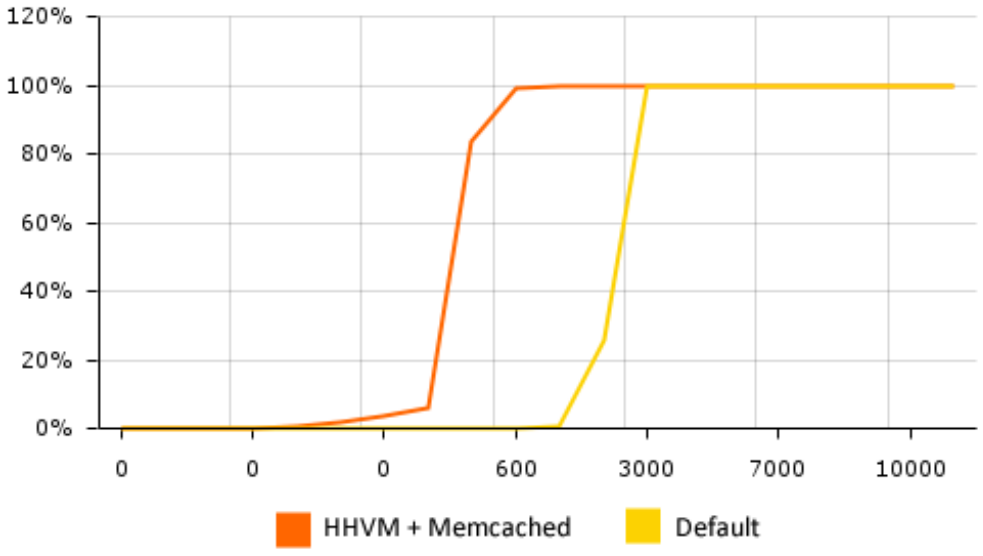


Figura 7.2: *Comparación entre ambos escenarios*

7.2.1.3. Average response time

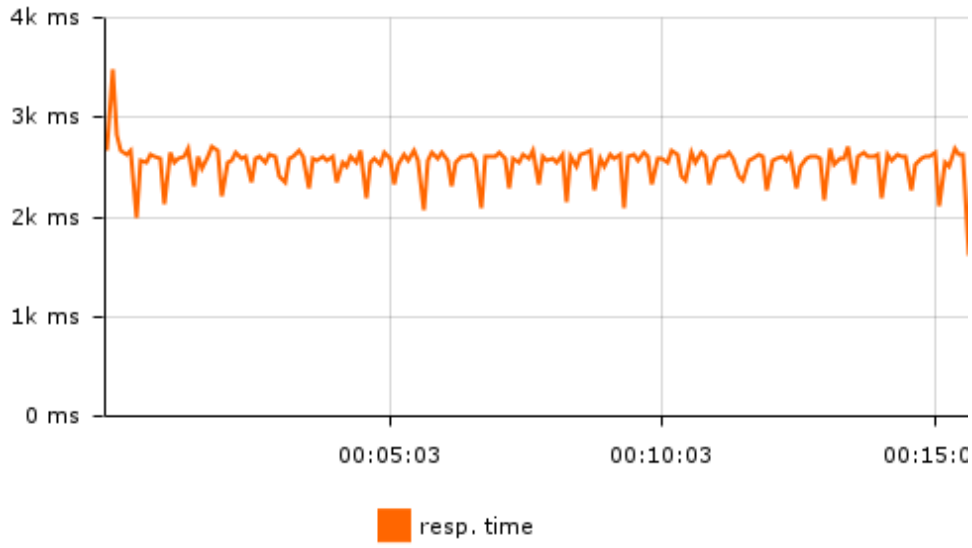


Figura 7.3: *Escenario Default*

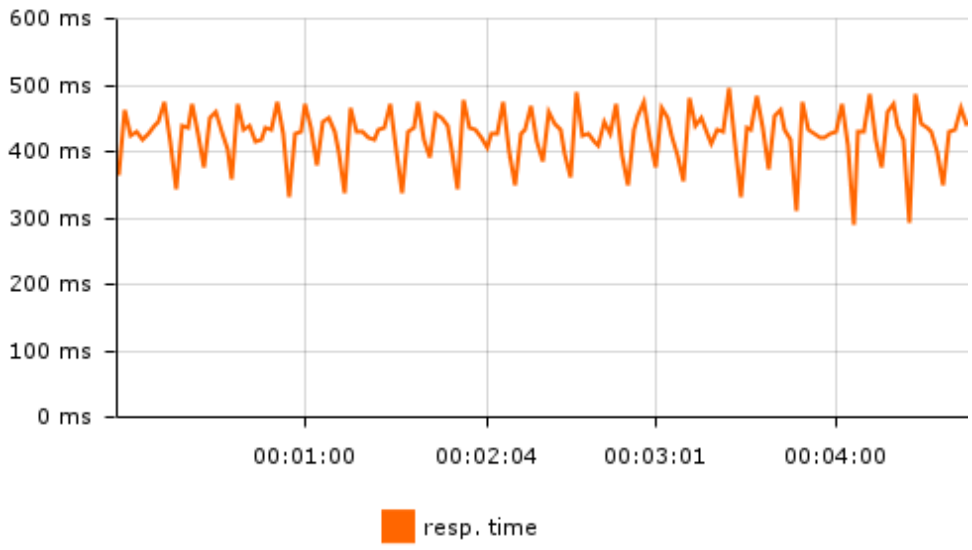


Figura 7.4: *Escenario HHVM con Memcached*

7.2.1.4. Response times vs Transactions per second

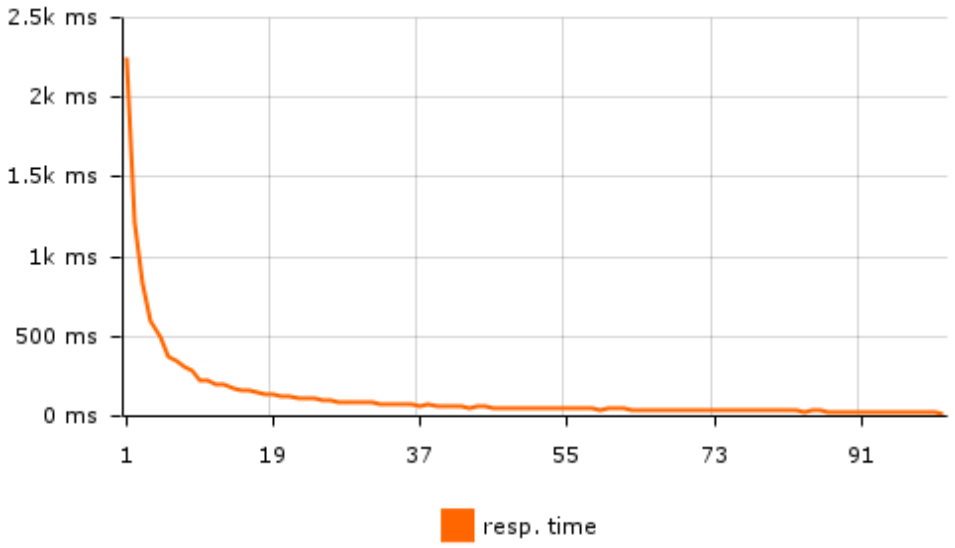


Figura 7.5: *Escenario Default*

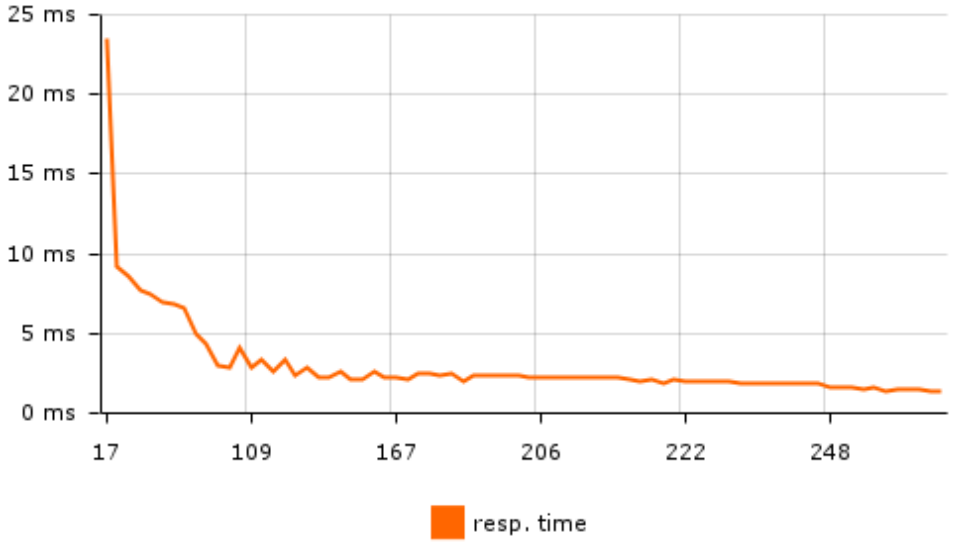


Figura 7.6: *Escenario HHVM con Memcached*

7.2.2. EventService::getAllByUserId

7.2.2.1. Response times distribution

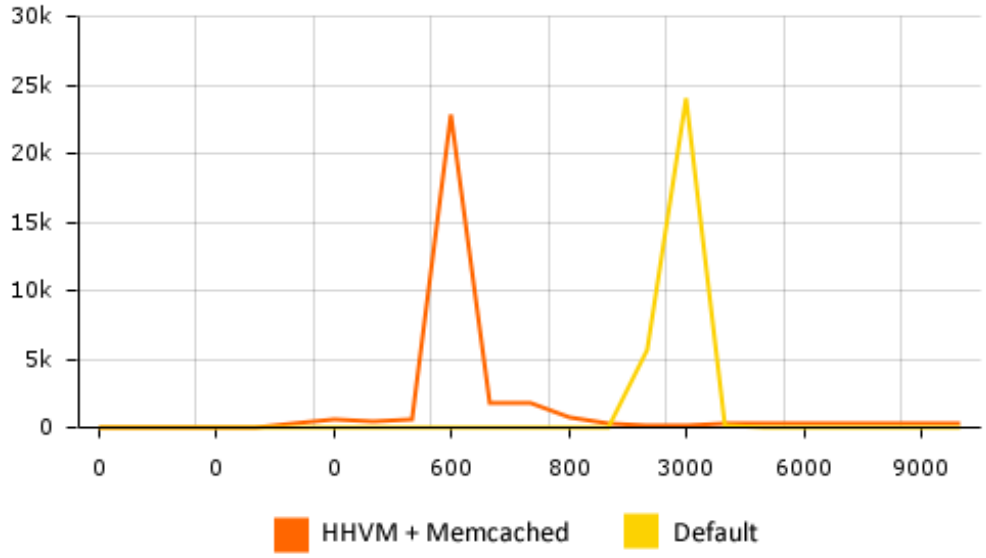


Figura 7.7: *Comparación entre ambos escenarios*

7.2.2.2. Response times percentiles

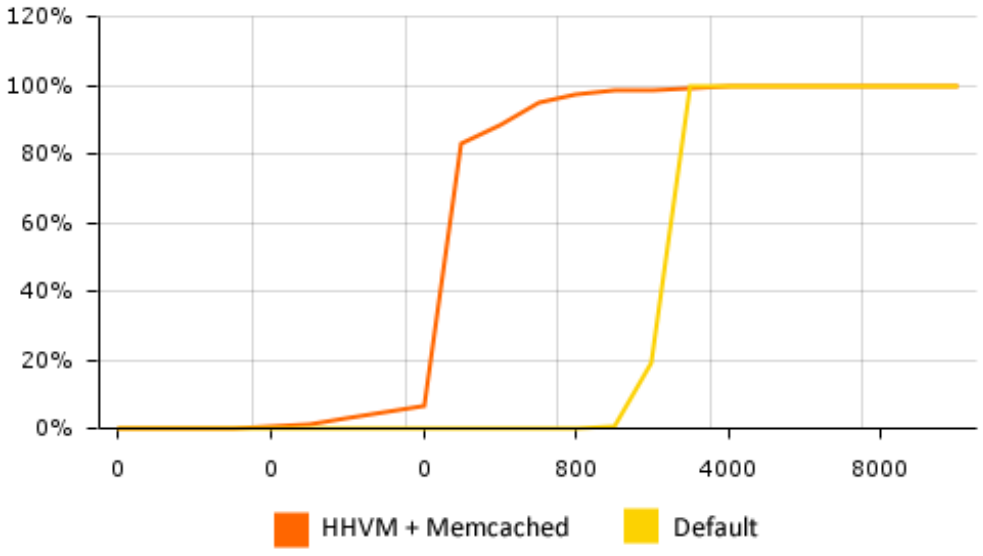


Figura 7.8: *Comparación entre ambos escenarios*

7.2.2.3. Average response time

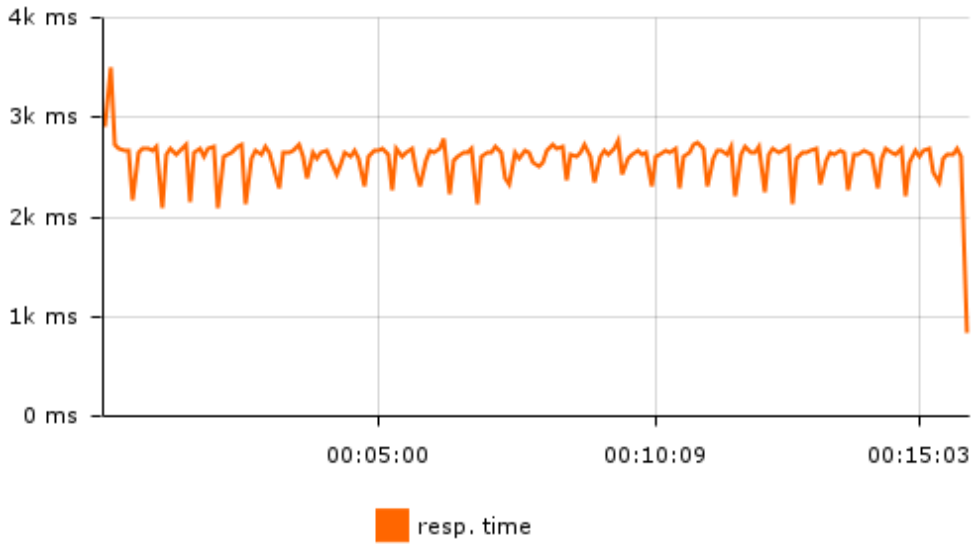


Figura 7.9: *Escenario Default*

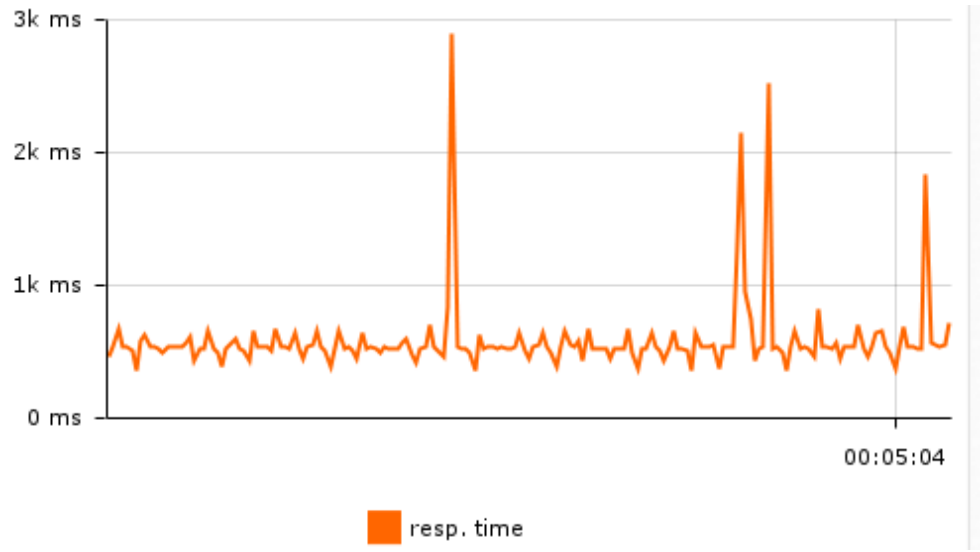


Figura 7.10: *Escenario HHVM con Memcached*

7.2.2.4. Response times vs Transactions per second

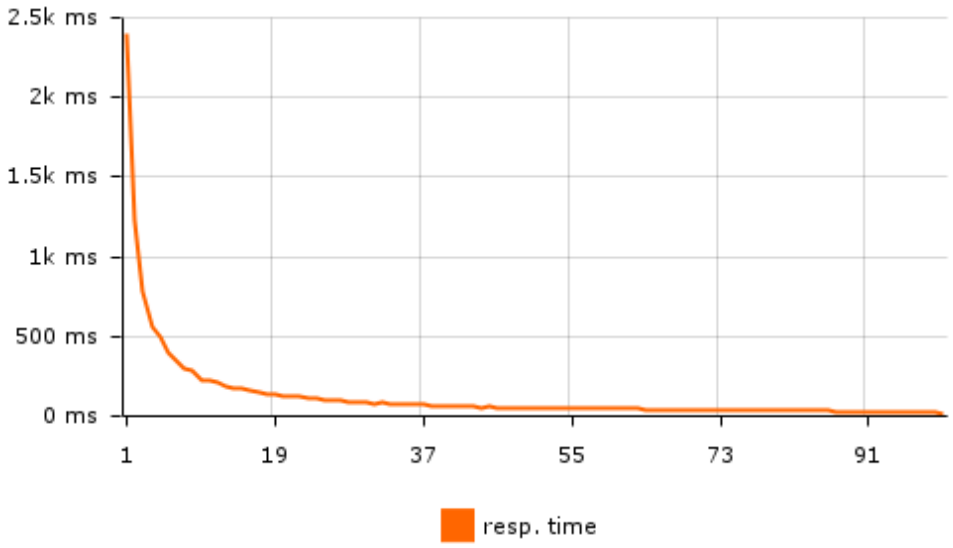


Figura 7.11: *Escenario Default*

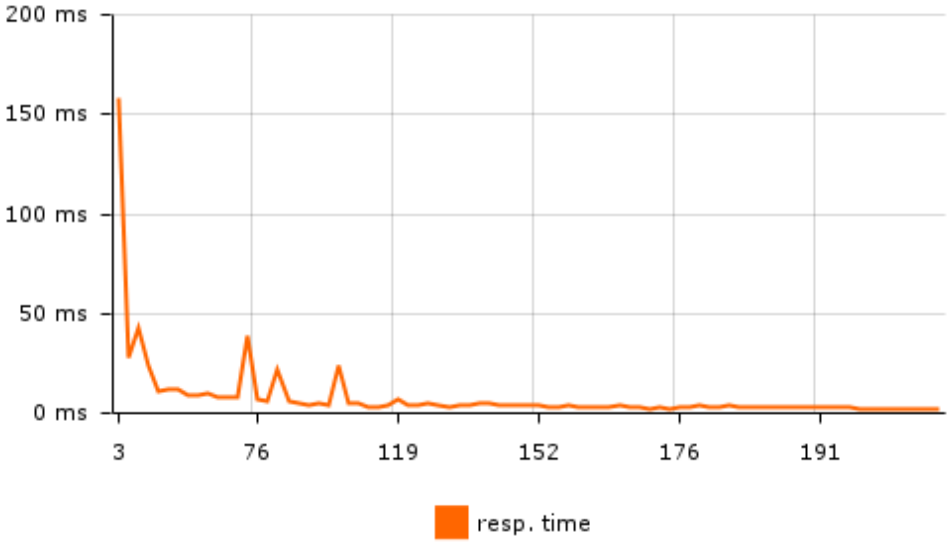


Figura 7.12: *Escenario HHVM con Memcached*

7.2.3. MarkerService::getByUserId

7.2.3.1. Response times distribution

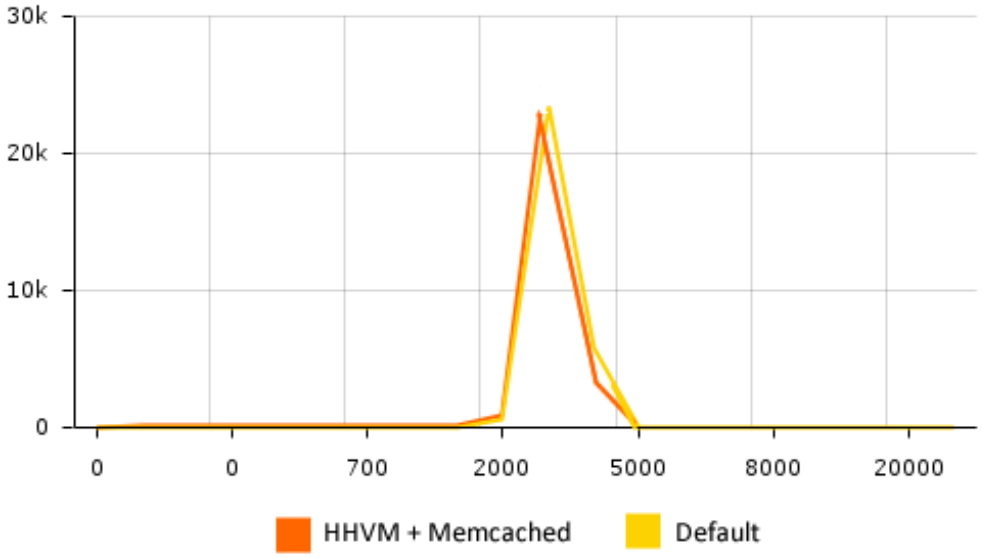


Figura 7.13: *Comparación entre ambos escenarios*

7.2.3.2. Response times percentiles

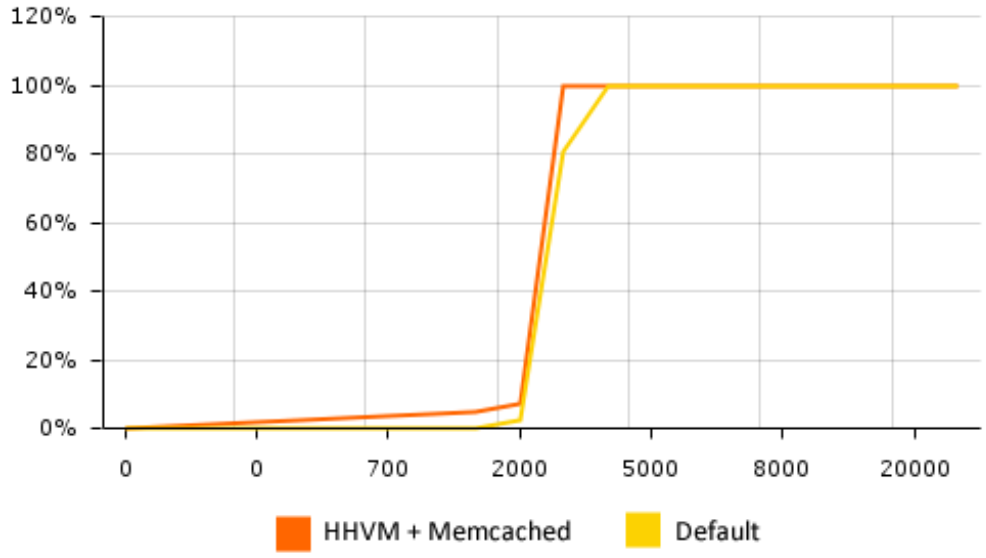


Figura 7.14: *Comparación entre ambos escenarios*

7.2.3.3. Average response time

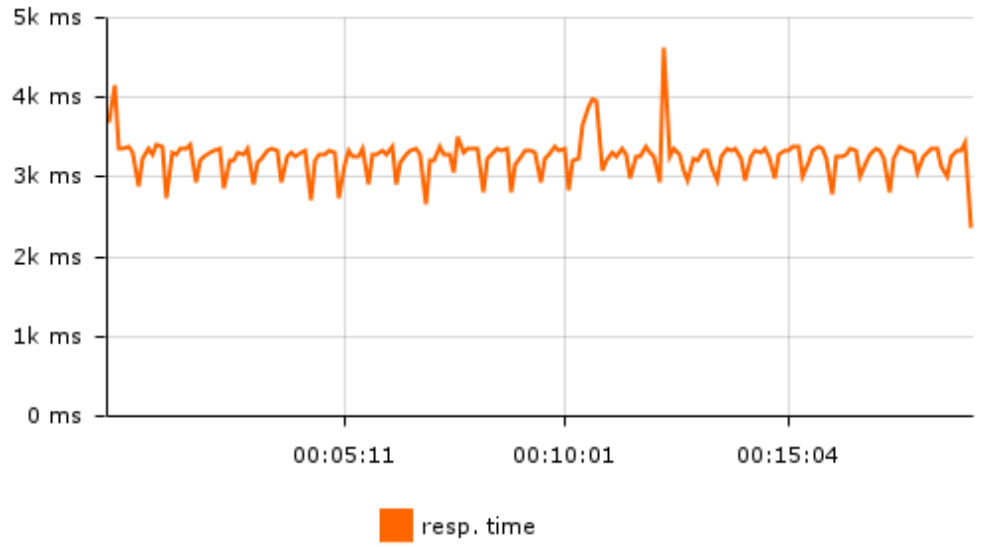


Figura 7.15: *Escenario Default*

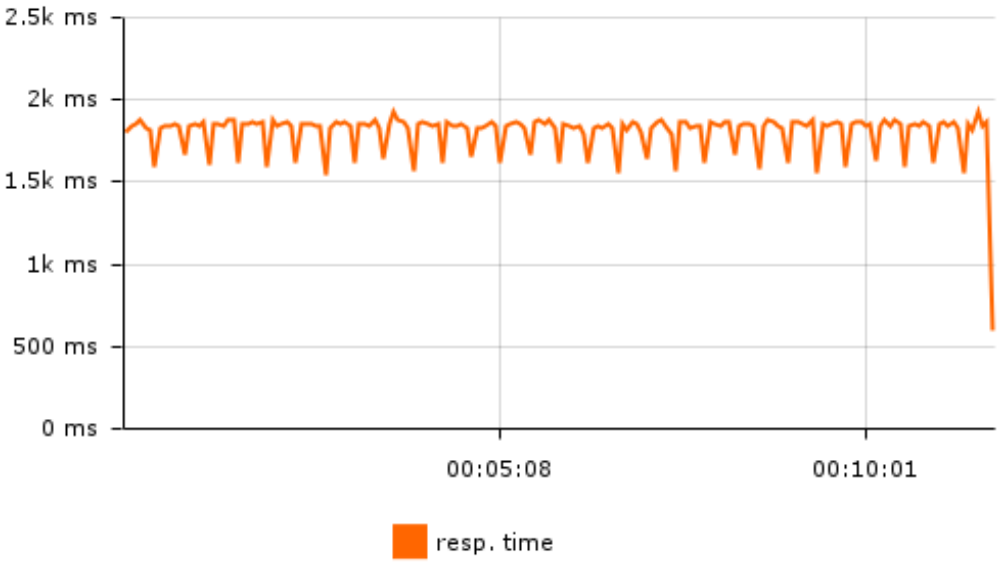


Figura 7.16: *Escenario HHVM con Memcached*

7.2.3.4. Response times vs Transactions per second

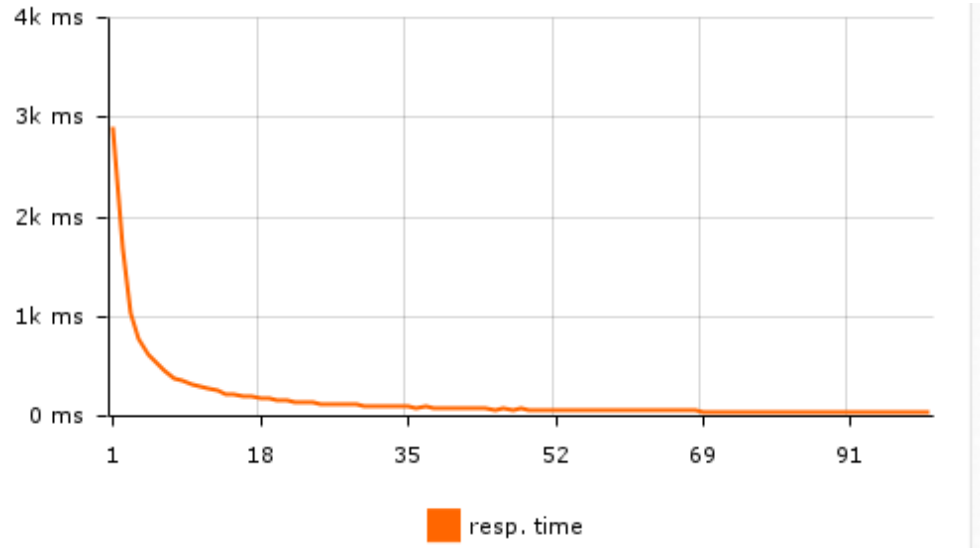


Figura 7.17: *Escenario Default*

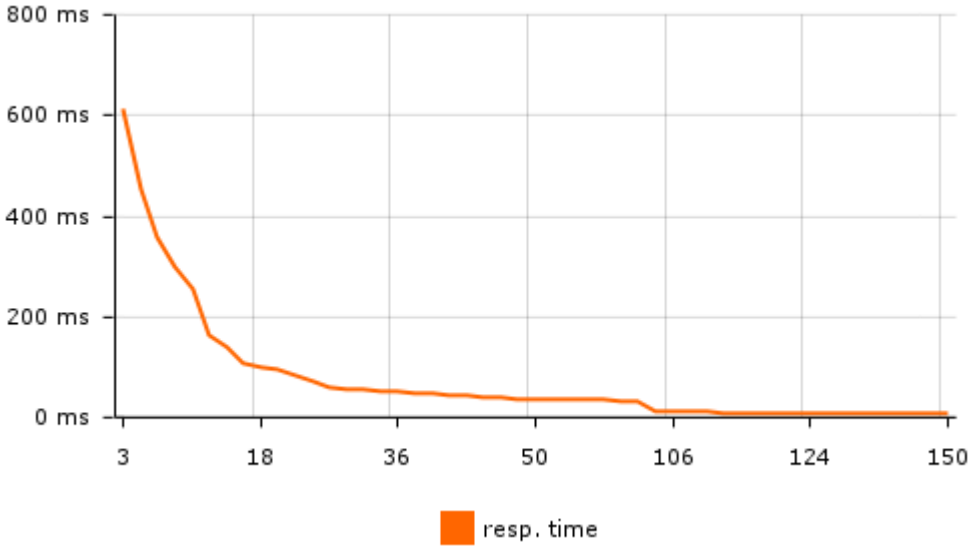


Figura 7.18: *Escenario HHVM con Memcached*

Capítulo 8

Conclusiones y Trabajo a Futuro

8.1. Análisis de objetivos

Con respecto a los objetivos propuestos para este trabajo, en el **Capítulo 2**, se despliega una implementación del backend de Placetribe, para poder crear distintos escenarios, y verificar cuanto es lo que se ha mejorado con respecto a un comienzo. Además se genera documentación que sirve como base para proyectos que deseen implementar estas herramientas.

Se estudia sobre los diversos conceptos involucrados para desarrollar el tema, como FastCGI, y se logran implementar.

A partir de los resultados exhibidos en el **Capítulo 7**, se puede concluir, que HHVM es la herramienta que mejoró en gran parte el rendimiento del servidor, mejorando el tiempo de respuesta hasta 15 veces en el caso de ChatService::getByUserId. Como se pudo apreciar, en todas las consultas seleccionadas, HHVM disminuyó por lo menos a un tercio del tiempo con respecto al escenario default.

En los distintos gráficos que exhiben en el **Capítulo 7**, se puede apreciar lo siguiente:

- Las peticiones se responden en tiempos relativamente cercanos.
- A mayor cantidad de transacciónes por segundo, menor es el tiempo en responder peticiones. Lo cual es razonable ya que, aquellas cantidades son inversamente proporcionales.
- HHVM, demostró ser una mejora en cada una de las consultas hechas. Variando bastante entre diferentes consultas.

Memcached por su lado, no mostró aportar con mejoras de velocidad, esto se debe a que estaba configu-

rado como un simple caché, función que se encuentra presente en los motores de bases de datos actuales, y que en este caso se desempeñaron mejor. No se descarta el uso de Memcached en un futuro, pero alomejor en situaciones, donde el caché que viene con los motores de bases de datos no sea suficiente.

Pese a las mejoras y beneficios que ofrece HHVM, hay que considerar su estatus de programa en fase de pruebas. Si se decide utilizar en un servidor para aplicaciones puntuales, no se ve un mayor problema. Pero si se decide utilizar para entregar un servicio para usuarios del tipo PaaS, es probable que el administrador se encuentre con bastantes problemas.

Finalmente, respecto al objetivo general "*Hacer un deployment de la plataforma backend de Placetribe, con tecnologías que mejoren el rendimiento del servidor*." se considera lograda según lo anteriormente expuesto.

8.2. Trabajo a Futuro

Como trabajo a futuro, se propone intentar otro tipo de optimizaciones, como utilizar *profilers*. Estos permiten analizar código y encontrar cuellos de botella, encontrar pedazos de código que son lentos y que se podrían mejorar.

Además se recomienda cambiar la librería con la que se interactúa con la base de datos a MySQLi, ya que MySQL se encuentra obsoleta en PHP 5.5.0, esta próxima a ser obsoleta en HHVM.

Placetribe esta programado usando el modelo MVC, pero internamente se comunica entre capas usando arreglos, para el envío de datos entre objetos. Podría considerarse y estudiar la conveniencia, de reescribir la capa de negocios utilizando bases de datos no relacionales, como MongoDB o NoSQL. Aunque hay casos en que pareciera no convenir, por la complejidad de algunas consultas.

También se recomienda, para una mejor compatibilidad de Placetribe con HHVM, reescribir la API REST de Placetribe. Ya que se han reportado problemas entre HHVM y mod_rewrite. La API REST de Placetribe hace uso intensivo de mod_rewrite, por lo que podría presentar problemas en el futuro y la solución del problema, por parte de los desarrolladores, parece no estar en un futuro cercano ya que no se encuentra en el roadmap próximo.

Referencias

[1] *Heroku Pricing*, Consultado el 29/12/2014 de:
`https://www.heroku.com/pricing`

[2] *Amazon EC2 Pricing*, Consultado el 29/12/2014 de:
`https://aws.amazon.com/es/ec2/pricing/`

[3] H. Zhao, I. Proctor, M. Yang, X. Qi, M. Williams, Q. Gao, G. Ottoni, A. Paroski, S. MacVicar, J. Evans and S. Tu. *"The HipHop Compiler for PHP"*. OOPSLA '12 Proceedings of the ACM, International conference on Object oriented programming systems languages and applications, Pages 575-586, ACM New York, NY, USA ©2012.

[4] Favre-Félix, Nicolas (16/02/2010). *"A review of PHP compilers and their outputs"*. Technow.owlient.eu. Consultado el 12/06/2014, de:
`http://technow.owlient.eu/index.php?post/2010/02/20/php-compilers`

[5] *"How do computer languages work?"*. Consultado el 29/12/2013, de:
`http://www.linux-tutorial.info/modules.php?name=Howto&pagename=Unix-and-Internet-Fundamentals-HOWTO/languages.html`

[6] *Github HHVM Readme.md*, Consultado el 07/01/2014 de:
`https://github.com/facebook/hhvm/`

[7] A. Bouaka, I. Puaut, and E. Rohou. *"Predictable Binary Code Cache: A First Step towards Reconciling Predictability and Just-in-Time Compilation"*, 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium. 2011

[8] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, Md. Wasi-ur-Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur and D. K. Panda, *"Memcached Design on High Performance RDMA Capable Interconnects"*, 2011 International Conference on Parallel Processing.

[9] *Phalanger*, Consultado el 29/12/2013 de:
http://www.php-compiler.net

[10] *"QCon 2012 HipHop Compiler Presentation"*. Consultado el 23/12/2013, de:
http://www.infoq.com/presentations/HipHop-Compiler-PHP

[11] *Speeding up PHP-based development with HHVM*. Consultado el 29/09/2014, de:
https://www.facebook.com/notes/facebook-engineering/speeding-up-php-based-development-with-hiphop-vm/10151170460698920

[12] HOFF, Todd. *"Facebook's Memcached Multiget Hole More Machines != more Capacity"*. Consultado el 29/09/14, de:
http://highscalability.com/blog/2009/10/26/facebooks-memcached-multiget-hole-more-machines-more-capacit.html

[13] *Hacking Hack on Heroku*. Consultado el 29/10/2014, de:
https://blog.heroku.com/archives/2014/3/21/hacking_hack_on_heroku

[14] *Heroku joins other PaaS providers in supporting PHP*. Consultado el 02/01/2015, de:
http://www.infoworld.com/article/2607983/paas/heroku-joins-other-paas-providers-in-supporting-php.html

[15] *Cloud Foundry Buildpack*. Consultado el 17/10/2014, de:
http://docs.cloudfoundry.com/docs/using/deploying-apps/buildpacks.html

[16] *Infographic: SMB Cloud Adoption Trends in 2014*. Consultado el 02/01/2015, de:
http://www.pcworld.com/article/2685792/infographic-smb-cloud-adoption-trends-in-2014.html

[17] *Enterprise-Class Memcached for Developers*. Consultado el 02/01/2015, de:
https://addons.heroku.com/memcachedcloud

[18] *Amazon ElastiCache*. Consultado el 02/01/2015, de:
http://aws.amazon.com/es/elasticache/

[19] *Memcached Cloud*. Consultado el 02/01/2015, de:
http://docs.run.pivotal.io/marketplace/services/memcachedcloud.html

[20] *HHVM on Heroku*. Consultado el 02/01/2015, de:
`http://hhvm.com/blog/1379/hhvm-on-heroku`

[21] *FasterCGI with HHVM*. Consultado el 05/10/2014, de:
`http://hhvm.com/blog/1817/fastercgi-with-hhvm`

[22] *Prebuilt packages on Centos 6.5*. Consultado el 04/09/2014, de:
`https://github.com/facebook/hhvm/wiki/Prebuilt-packages-on-Centos-6.5`

[23] *Apache Core Features: ServerTokens*. Consultado el 07/09/2014, de:
`http://httpd.apache.org/docs/2.2/mod/core.html#servertokens`

[24] *Apache Core Features: AllowOverride*. Consultado el 07/09/2014, de:
`http://httpd.apache.org/docs/2.2/mod/core.html#AllowOverride`

[25] *FastCGI — Facebook HHVM Documentation*. Consultado el 07/09/2014, de:
`https://github.com/facebook/hhvm/wiki/FastCGI`

[26] *How to use Memcache with PHP*. Consultado el 18/10/2014, de:
`http://www.script-tutorials.com/how-to-use-memcache-with-php/`

[27] *The Memcache class*. Consultado el 18/10/2014, de:
`http://php.net/manual/en/class.memcache.php`

[28] DEVORE, Jay L. –6ta edición– *"Probabilidades y Estadísticas para Ingeniería y Ciencias"*.