Alternativas para la escalabilidad de aplicaciones en plataformas web de alta concurrencia*

Jorge Luis Irey Núñez

Resumen

En la actualidad, la denominada "Internet de las cosas" ha convertido en indispensables las aplicaciones desplegadas en Internet: los equipos que actúan como clientes no son exclusivamente PC o laptops, sino también teléfonos inteligentes, tabletas y en general diversos tipos de dispositivos (televisores, por ejemplo) que ejecutan aplicaciones. Esta caracterización de "indispensable" se asocia básicamente con Requerimientos No Funcionales (RNF) como disponibilidad, escalabilidad y tolerancia a fallos. Este artículo explora, desde un punto de vista técnico, un nuevo paradigma de programación que permite construir las llamadas aplicaciones web "reactivas" las cuales poseen intrínsecamente las características de manejo por eventos, escalabilidad, resiliencia y sensibilidad en contraste con las aplicaciones web tradicionales.

Palabras clave: requerimientos no funcionales, escalabilidad, aplicaciones reactivas

Alternatives to the scalability of applications in high concurrency web platforms

Summary

At present, the so-called "Internet of Things" has turned indispensable the applications deployed on the Internet; the computers that act as clients are not exclusively PC's or laptops, but also smart phones, tablets, and in general many different types of devices that run applications. This characterization of indispensable is basically associated with non-functional requirements (NFR), such as availability, scalability, and fault tolerance. This article explores from a technical point of view, a new programming paradigm that lets you build the so called reactive web applications which inherently have the event-handling characteristics, scalability, resilience and sensitivity in contrast to traditional web applications.

Key words: non-functional requirements, scalability, reactive applications

* El autor expresa su agradecimiento al alumno Luis Alejandro Ruiz del Águila, por el esfuerzo dedicado en preparar el ambiente de pruebas, y al Círculo de Estudios, Investigación y Desarrollo de Software (CEIDS), por proporcionar el *hardware* requerido para las pruebas.

Introducción

Desde los inicios de Internet el mayor problema con el que se han enfrentado los administradores de los sitios web ha sido la aleatoria concurrencia de usuarios, que en situaciones especiales ocasionaban "picos" de demanda y sobrepasaban la capacidad del *hardware* donde estaban ejecutándose las aplicaciones web. En algunos casos podían tratarse de ataques para denegación de servicio (DoS) y en otros la concurrencia causada por la publicación de algún contenido importante (efecto *slash dot*); ejemplos cercanos de esto último fueron la aplicación para el registro de devolución de aportes al Fonavi, que en su primer día de operación colapsó, al igual que la aplicación del Seguro Social de los Estados Unidos.

Kegel (1999) manifestaba: "Es hora de que los servidores de Internet puedan manejar diez mil clientes al mismo tiempo, ¿no crees? Después de todo, la web es un lugar muy grande ahora y las computadoras también". Kounev y Buchmann (2003) reconocen las dificultades que enfrentan los responsables de la instalación de las aplicaciones en Internet para garantizar los niveles de servicio requerido, y tomando como base el *benchmark* SpecjAppServer2002 plantean el uso de modelos analíticos para mejorar el planeamiento de la capacidad. De igual forma, Kounev y Buchmann (2003) muestran cómo los modelos de Petri y las colas pueden ser empleados para analizar el desempeño de sistemas distribuidos para *e-business*. Los autores acotan que mientras los modelos basados en redes de colas permiten modelar la contención de recursos y la programación de tareas, no son adecuados para representar el bloqueo y la sincronización de procesos para lo cual son apropiadas las redes de Petri. En base a ello, proponen un modelo mixto y lo aplican sobre el *benchmark* SPECjAppServer2001.

Broadwell (2004) señaló cómo se pueden emplear las métricas de performance, como el tiempo de respuesta, cuando se realizan *benchmarks* de confiabilidad en sistemas denominados "en línea". El estudio apunta a analizar básicamente todas las métricas que consideran cómo una falla puede afectar el desempeño del sistema y a dichas métricas les atribuye el nombre de *performability metrics*. Kounev (2005) realizó una revisión de los criterios tomados para definir el nuevo *benchmark* de servidores de aplicaciones SpecjAppServer2004, y ofrece una explicación de las métricas y sus posibilidades de aplicación. Liotopoulos (2005) analizó los desafíos y factores que afectan el desempeño de un sistema de votación electrónica empleando para ello un modelo cerrado de redes y resolviéndolo mediante la técnica de Análisis de valor Medio exacto (EMVA); mientras que Kambhampaty y Srinivas (2007) sugieren una aproximación para modelar el desempeño de aplicaciones J2EE y .NET durante las fases tempranas del desarrollo y evitar así problemas posteriores que puedan ser detectados en la puesta en producción de la aplicación. Con el avance tecnológico actual, el *hardware* de las computadoras

no constituye el cuello de botella. Sin embargo, se enfrenta el dilema de calcular la capacidad necesaria para atender probables picos de demanda. Ante esto, normalmente se opta por algunas soluciones:

a) Escalar horizontalmente, agregando más equipos de *hardware* y desplegando las aplicaciones en cada uno de ellos.
b) Escalar verticalmente, cambiando la infraestructura hacia un *hardware* más potente de lo necesario.
c) Emplear infraestructura de nube (*cloud computing*), que permite un consumo de recursos de manera elástica.

Dichos escenarios son independientes de la plataforma de desarrollo de *software* que se utilice y es muy probable que en este ejercicio de planeamiento de capacidad (*capacity planning*) muchas veces se olvide que las aplicaciones se ejecutan sobre una pila de *software* conformada básicamente por un programa denominado "servidor de aplicaciones" y este, a su vez, se ejecuta sobre otro programa llamado "sistema operativo". En esta situación ¿es posible mejorar dramáticamente la escalabilidad de una aplicación sin cambiar necesariamente el *hardware* sobre el que se ejecuta?

El presente artículo se enfoca sobre la tecnología Java EE y el ecosistema que tiene alrededor para evaluar temas de escalabilidad, considerando que las aplicaciones "modernas" pueden clasificarse, según Layka (2014), en uno o varios criterios como responsivas, *single-page*, *real-time* y reactivas, como se aprecia en la tabla 1.

Tabla 1. Tipo de aplicación y sus características

Tipo de aplicación	Características	Tecnologías
Responsivas	Se adaptan al tamaño de la pantalla del dispositivo sobre el cual se ejecutan (*device agnostic*). Aquí aplica un concepto conocido como *mobile-first*, en el cual las pantallas de la aplicación deben diseñarse basados en tamaños pequeños de pantalla para luego ir expandiendo el tamaño hasta lograr pantallas de computadores personales.	✓ Tecnologías del lado cliente (*from end*): CSS3, jQuery, LESS, CoffeeScript ✓ *Frameworks* como: Bootstrap (el más conocido), Pollyfills, Modernizr
Single page (SPA)	La aplicación consta de una única página, que es cargada en el navegador de tal forma que nunca se invoca ni se pasa el control a otra página. El resto de componentes (imágenes y/o textos) aparecen o desaparecen en respuesta a eventos.	Node.js Angular.js

Real time	Son aplicaciones que entregan una respuesta a eventos de manera casi instantánea pero en forma asíncrona y bidireccional entre el navegador y el servidor.	WebSockets
Reactivas	Son aplicaciones diseñadas para estar siempre listas a recibir eventos y responder a ellos escalando ante concurrencias fortuitas, recuperándose ante fallos inesperados y brindando una experiencia de usuario aceptable.	Play2 Framework Scala, Akka

Fuente: Layka, 2014.

1. Requisitos no funcionales

La ingeniería de requerimientos es un conjunto de actividades para descubrir, documentar y mantener un conjunto de requisitos que deben ser implementados por el *software* por construir. Esta disciplina busca que los requisitos sean verificables, no ambiguos y a la vez cuantificables (que se puedan medir) en lo posible. Sin embargo, teniendo en cuenta que solo con los requisitos funcionales (los cuales definen los aspectos de negocio que se desea cubrir) se tiene un gran trabajo, se puede justificar que generalmente se deje de lado los requisitos No funcionales, los cuales jugarán un papel importante durante el despliegue y funcionamiento de las aplicaciones en internet.

Tradicionalmente se desea que las aplicaciones web cumplan ciertos requerimientos no funcionales, los cuales en algunos casos pueden ser contrapuestos:

- Disponibilidad (*availability*). Es el porcentaje de tiempo que el sistema está operativo. El principal parámetro para medir la disponibilidad es el tiempo medio entre fallos (MTBF), pero hay que considerar también el tiempo de recuperación. En sitios web de alta concurrencia, la disponibilidad de los servicios brindados es absolutamente fundamental para los visitantes. Si el sitio web no está disponible no solo genera problemas de imagen a la empresa, sino también podrían ocurrir pérdidas monetarias para ambas partes, por lo que se deben diseñar las aplicaciones para estar disponibles el mayor periodo de tiempo posible y a la vez considerar que sean resistentes ante errores. La alta disponibilidad en sistemas distribuidos requiere el diseño detallado de la redundancia de componentes clave, la recuperación rápida (o automática) en caso de fallos parciales del sistema y la degradación progresiva cuando se producen problemas.

- Tolerancia a fallos (*fault tolerance*). Aun con una alta disponibilidad, una falla puede ocasionar problemas. Es decir, la disponibilidad no garantiza por sí sola la continuidad del servicio de forma transparente. La tolerancia a fallos expresa la capacidad del sistema para seguir operando correctamente (o por lo menos informárselo amigablemente al usuario) ante una falla en alguno de sus componentes. Por lo tanto, la tolerancia a fallos implica detectar la falla y continuar operando el servicio de forma transparente para la aplicación.
- Tiempo de respuesta (*response time*). La velocidad del sitio web afecta el uso y la satisfacción del usuario. Por ello se consideran dos maneras de mejorar el tiempo de respuesta y la experiencia de uso del usuario: diseñar las aplicaciones para obtener respuestas rápidas o delegar en la infraestructura el monitoreo del tiempo de respuesta y el ajuste automático de los componentes involucrados para mantener el nivel de servicio (SLA).
- Escalabilidad (*scalability*). Se define como la capacidad del sistema para crecer ante un incremento en la concurrencia de usuarios, sin aumentar su complejidad ni disminuir su rendimiento. El crecimiento de un sistema distribuido puede introducir cuellos de botella y latencias que degradan su rendimiento (por ejemplo la coordinación entre mayor número de nodos). La escalabilidad puede referirse a diversos parámetros del sistema: cantidad de tráfico de red adicional que se puede manejar, facilidad para agregar más capacidad de almacenamiento, cantidad de transacciones adicionales que se pueden procesar, etcétera.
- Manejabilidad (*manageability*). Se debe tener en cuenta la facilidad de diagnóstico y la comprensión de los problemas cuando se producen, facilidad de realizar cambios o modificaciones (instalación de cambios en producción) y sobre todo tener el control de lo que está ocurriendo en el sistema (monitoreo).
- Costo. Sin considerar el costo de desarrollar una aplicación, debe estimarse el costo de operar el sistema (facturación): consumo de CPU, consumo de ancho de banda, espacio de almacenamiento, etcétera.

2. Escalando en la web

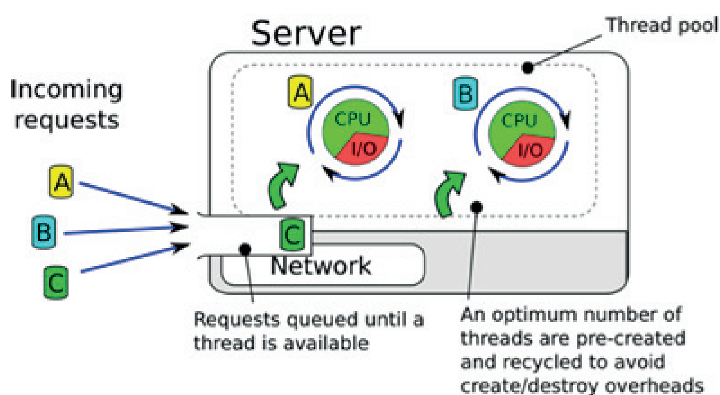
Generalmente, en las aplicaciones web el diseño se centra únicamente en lo que el sistema debe hacer cuando se presiona un clic mientras que la arquitectura se enfoca en lo que el sistema debe hacer cuando cientos de usuarios presionan el mismo clic de manera simultánea.

En Java EE lo comúnmente utilizado es que los contenedores web dentro de los servidores de aplicaciones empleen un *thread* (hilo de ejecución) por cada petición (*request*) de un cliente. Esta práctica conlleva a que bajo situaciones de alta concurrencia el contenedor web deba generar gran cantidad de *threads* para

atender las peticiones, generando limitaciones de escalabilidad por problemas de memoria o por agotar la capacidad de *threads* disponibles.

En la figura 1 se muestra un esquema del funcionamiento de un contenedor web genérico: se define un número "n" de *threads* para atender peticiones HTTP, lo cual significa que el servidor es capaz de procesar "n" peticiones concurrentes. Las peticiones que no alcanzan algún *thread* de ejecución deben esperar en una cola hasta que alguno de ellos termine su trabajo y quede en estado disponible. Esta implementación fue ampliamente estudiada por Menascé (2001), mediante simulación de colas y redes de Petri.

Figura 1. Arquitectura *Thread-pool*

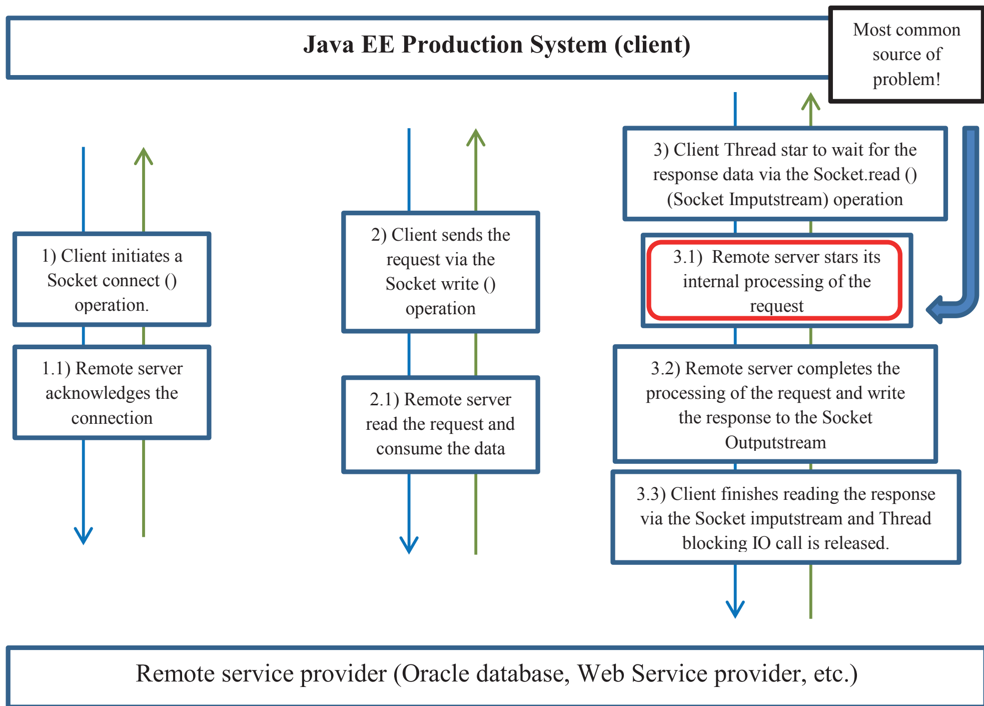


Fuente: Taing, 2011.

En la figura 2 se muestra la interacción de una petición HTTP dentro de un servidor JEE estándar (pasos 1 y 2), el procesamiento (paso 3) marcado con la flecha y la respuesta.

Para evitar las limitaciones de escalabilidad, debería asegurarse que siempre existan *threads* disponibles para atender nuevas peticiones de los clientes y para ello deberían evitarse situaciones en las cuales el *thread* para poder completar su trabajo espera por un recurso o por la ocurrencia de un evento: este tipo de operación se denomina *blocking operation* (el paso 3.1 de la figura 2 marcada por la flecha).

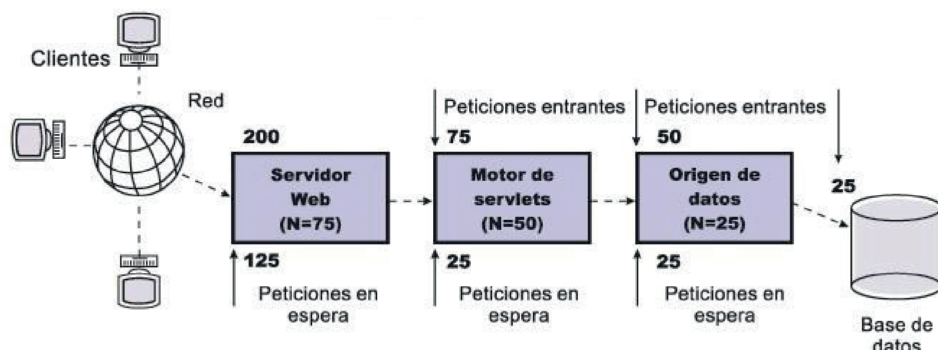
Figura 2. Interacción de un *thread* de ejecución



Fuente: Charbonneau, 2011.

Por ello una arquitectura web típica posee la configuración mostrada en la figura 3, donde a manera de ejemplo se grafica que si llegan 200 conexiones de clientes HTTP, el servidor web procesa 75 (N = número de *threads* de ejecución configurados) y coloca en espera a 125 (la diferencia). Los 75 *threads* ejecutan tareas (del tipo *blocking operation*) que pasan hacia el *Servlet Engine*, el cual solo soporta 50 concurrentes, dejando 25 en cola y así sucesivamente van pasando las peticiones hacia la parte interna de la plataforma hasta llegar a la base de datos o a sistemas internos de la empresa.

Figura 3. Esquema típico de una plataforma web



Fuente: IBM Knowledge Center, 2011.

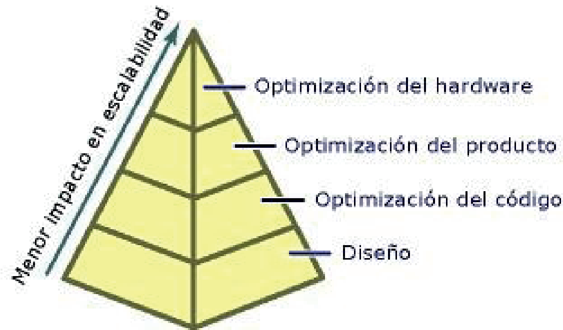
La descripción mencionada de la figura 3 se conoce como efecto embudo (*funnel effect*), en el cual la plataforma está preparada para autorregular la concurrencia de usuarios y donde las peticiones que no llegan a quedar en la cola del Web Server se encolan en el nivel de red (por lo que el usuario percibe una lentitud en la conexión a una página web). El tiempo total de respuesta que experimente el usuario está dado por el viaje redondo (ida y vuelta) que haga su petición HTTP desde que sale de su dispositivo cliente hasta que regresa luego de cruzar por todas las capas de la plataforma mostrada. Por ello cuando ocurren saturaciones en las capas internas de la plataforma, el efecto embudo se propaga hacia afuera generando saturación y sobrecarga (por desbordamiento de la cola de espera).

La escalabilidad debe formar parte del proceso de desarrollo porque esta no es una característica separada que se pueda agregar después durante la fase de "estabilización" del proyecto: las decisiones que se tomen durante las primeras fases de diseño y codificación determinarán en gran medida la escalabilidad de la aplicación, como se aprecia en la figura 4.

Las aplicaciones modernas exigen tiempos de respuesta más que aceptables, por lo cual la tecnología Java EE ha evolucionado en el tiempo (desde J2EE hasta Java EE7) y ha propuesto el uso del procesamiento asíncrono, que permite asignar justamente las *blocking operations* a nuevos *threads* de tal forma que el *thread* original se libere y pueda atender nuevas peticiones que llegan al contenedor web.

Con la irrupción de aplicaciones en la nube (*cloud*), se recomienda que el diseño de las aplicaciones contemple aspectos relacionados con recuperación ante fallas y la arquitectura contemple aspectos relacionados con la resiliencia del sistema.

Figura 4. Pirámide de escalabilidad



Fuente: Microsoft Developer Network, 2014.

3. Las aplicaciones reactivas al rescate

El manifiesto reactivo (http://www.reactivemanifesto.org/) explica detalladamente el motivo por el que se denominan aplicaciones reactivas. Este manifiesto propone una nueva arquitectura, que permite construir aplicaciones que son manejadas por eventos, escalables, resilientes, a la vez que son responsivas, lo cual permite desplegarlas de manera tradicional en infraestructura propia o de manera más elástica, en la nube.

Proactivo y reactivo son conceptos asociados generalmente a situaciones laborales donde la proactividad es una característica deseable del trabajador. Pero en el área de desarrollo de sistemas ¿es poco deseable ser reactivo? Según la definición del *Diccionario de la lengua española* (RAE, 2014) "reactivo" es un adjetivo utilizado más como sustantivo masculino que significa "que produce reacción", lo cual dentro del desarrollo de sistemas se entiende como una aplicación que está lista para reaccionar ante estímulos externos, los cuales dentro del contexto del manifiesto reactivo son básicamente cuatro:

a) Reaccionar ante eventos, lo cual permite que las otras tres características funcionen. La arquitectura que propone es el uso de comunicaciones asíncronas y operaciones de tipo *non-blocking*, las cuales habilitan que las diferentes capas de la aplicación funcionen independientemente ("bajo acoplamiento" o *loose-coupling*) de la forma en que el evento se propaga y dado que bajo este esquema las capas "duermen" hasta la ocurrencia de un evento, el uso de los recursos es más eficiente.

b) Reaccionar ante concurrencia, lo cual se enfoca en la escalabilidad de la aplicación. Una aplicación escalable se expande por demanda sin

necesidad de rediseñar ni reescribir el código, siendo capaz de permitir el escalamiento vertical y horizontal, así como soportar el incremento (*scale out*) y la reducción de recursos (*scale in*) de forma dinámica. Esto se logra diseñando la aplicación de tal manera que no mantenga estado (*state-less*) entre la comunicación del cliente con el lado servidor permitiendo la transparencia de ubicación a nivel de red (no importa dónde esté ubicado el recurso o subsistema).

c) Reaccionar ante fallas, lo cual significa construir aplicaciones resilientes que se recuperen ante errores en las diferentes capas de la aplicación. La técnica tradicionalmente costosa para evitar fallas e interrupciones de servicio ha sido siempre generar *clusters* de servidores. El manifiesto reactivo propone el concepto de "resiliencia", la cual se puede entender como la capacidad que tiene el sistema para sobreponerse ante errores o fallas: esto se implementa mediante el aislamiento entre los componentes de la aplicación.

d) Reaccionar ante los usuarios, manteniendo el tiempo de respuesta sin interesar la carga que soporte el sistema, presentando a la vez una interface de usuario adaptativa a los diferentes dispositivos. El "Manifiesto Reactivo" claramente indica que esto no está referido al concepto de *Responsive Design* sino más bien a interfaces de usuario en tiempo real que presentan una experiencia enriquecida para el usuario.

4. Caso de análisis

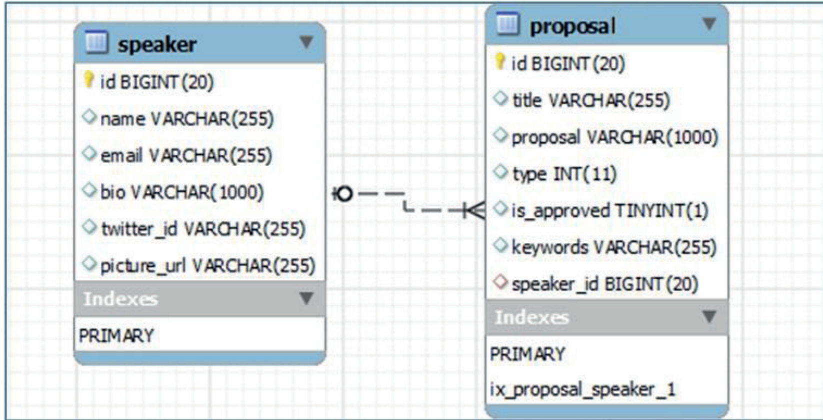
Tomando como base la aplicación ficticia desarrollada en el video tutorial de TypeSafe, se tiene el caso de una aplicación que simula un sitio de conferencias y tiene básicamente dos funciones:

a) Aceptar el registro de conferencistas con los datos de la ponencia (los URI que maneja son "/new" y "/submit").

b) Mostrar en la página principal (*landing page, home page* o página de bienvenida) la información del ponente registrado con estado "1" (el URL que maneja es "/").

El esquema de la base de datos tiene dos tablas. La tabla *speaker* contiene la información del conferencista y la tabla *proposal* contiene la información de la ponencia. La relación se considera de tipo One-to-One aunque la herramienta de Ingeniería Reversa la modele como One-To-Many.

Figura 5. Tablas *speaker y proposal*



Elaboración propia.

La lógica de la página de bienvenida es: llega la petición HTTP y la aplicación debe consultar a la base de datos para buscar un único registro en la tabla *proposal* que tenga *type* igual a cero. Si se encuentra más de un registro o existe algún problema con la base de datos, el sistema debe indicarle al usuario el mensaje de *Cooming soon*¡. En caso contrario se muestra la información obtenida.

El visitante puede registrar una propuesta (enlace a "/new") y llenar los campos del formulario que se muestra. Luego presiona el botón de envío (enlace a "/submit") y la información debe grabarse en la base de datos y el navegador debe regresar a la página de bienvenida.

Para tener una aplicación base se codificaron las lecciones de la 1 a la 4 del tutorial de TypeSafe Activator, el cual está basado en el uso de *Play framework for java*. Un fragmento del controlador se muestra en el siguiente extracto:

```
public class Application extends Controller {

  private static Form<Proposal> proposal Form = Form. form (Proposal. class );

  public static Promise<Result> index(){
     Promise<Propasal> keynote = Proposal. findkeynote ();

     Promi se<Result> result = keynote.map ( new Function< Proposal,Result>() {

          @Override
          public Result apply(Proposal keynote) throws Throwable {
             return ok (views.html.index.render(keynote));
          }
     }};
     return result;
  }
```

Extracto donde se puede apreciar que el trabajo asíncrono está dado por el uso de las clases "Promise" proporcionada por el *framework*, similar al *fork-join* de Java 7. Luego, se codificó empleando Java EE, como se aprecia a continuación:

```
public class IndexServlet extends HttpServlet {

    @Override
    protected void doGet (HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        // Buscar el Keynote y devolver Respuesta implementando:
        // find.where().eq("type",SessionType.Keynote). findUnique();

        HttpSession ses = request.getSession ( true );

        EntityManagerFactory emf =
        Persistence. createEntityManagerFactory ("playconfServletPU");
        EntityManager em = emf.createEntityManager();
        Proposal p = null ;

        try {
            Query q = em.createQuery ("SELECT p FROM Proposal p WHERE p.type = ?1" );
            q.setParameter(1,SessionType. KeyNote );
            p = (Proposal) q.getSingleResult();
        } catch (Exception e) {
            // Si ocurre un error debe usarse esto
            P = new Proposal ();
            p.setTitle ("COMING SOON!" );
            p.setProposal("");
            Speaker speaker = new Speaker ();
            speaker.setName ("");
            speaker.setPictureUrl("");
            speaker.setTwitterId( "");
            p.setSpeaker(speaker);
        } finally {
            //Colocar en la session el valor
            ses.setAttribute( "propo" ,p);
        }

        RequestDispatcher rd = request.getRequestDispatcher( "/inicio.jsp" );
        rd.forward (request, response);
```

Nótense las diferencias respecto a la forma de programar: no se está empleando las funcionalidades asíncronas de los Servlets. En el caso de la aplicación base algunos recursos son compilados automáticamente (hojas LESS, Coffee Script y SASS), mientras que en el caso de JavaEE se tuvo que emplear las siguientes herramientas en línea para generar los archivos para ser colocados en la aplicación:

a) Online LESS Compiler, para compilar las Hojas de estilo LESS (http://winless.org/online-less-compiler).

b) Try CoffeeScript Online, para compilar los JavaScript desarrollados con Coffee Script (http://www.compileonline.com/try_coffeescript_online.php).

c) Sass to CSS, para convertir de SASS a CSS (http://sasstocss.appspot.com/).

El cuadro comparativo de las tecnologías empleadas en cada versión de la aplicación se muestra en la tabla 2.

Tabla 2. Comparación de versiones

	Versión "Tradicional"	Versión "Reactiva"
JVM	Basado en la JVM de Java 1.7.0_67	Basado en la JVM de Java 1.7.0_67
Controlador	Servlet escrito en Java	Controller escrito en Play Framework for Java
Vista	JSP/JSTL / HTML5 / CSS3	HTML5 / CSS3 (CoffeScript, LESS y SASS)
Modelo	JPA EclipseLink 2.1	eBean (aunque también soporta JPA)
Base de datos	MySQL 5.6.21	MySQL 5.6.21
Servidor de aplicaciones	GlassFish 4.0	Netty embebido dentro de TypeSafe Activator

Elaboración propia.

Se debe aclarar que este escenario enfrenta dos posiciones respecto a los servidores de aplicaciones: la tradicional, en la que un servidor de aplicaciones puede ejecutar múltiples aplicaciones, las cuales deben configurarse y desplegarse separadamente, generando un entorno en el cual el monitoreo individual de las aplicaciones es bastante complicado, siendo este el esquema que ha prevalecido desde los inicios de Java. Por otro lado, están los esquemas de despliegue más "modernos", en los cuales el servidor de aplicaciones se encuentra embebido con la aplicación, lo que permite un despliegue, monitoreo y control más eficiente de los recursos.

La configuración de la conexión a la base de datos es diferente en cada servidor, pues mientras que GlassFish emplea el "JDBC Connection Pool", en el caso de TypeSafe Activator se usa un *pooling* particular, en consecuencia:

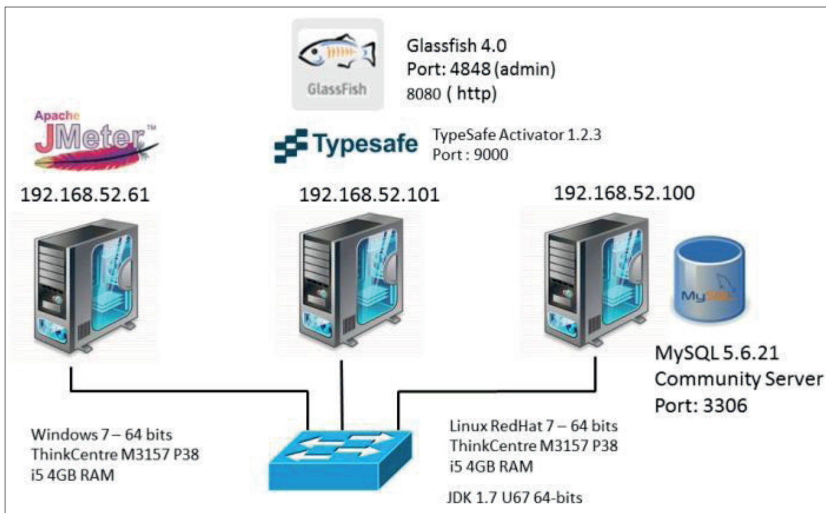
a. Para el caso de GlassFish, se definió un Data Source que tiene por debajo un *pool* de conexiones JDBC apuntando al servidor de base de datos MySQL. Durante las pruebas se modificó el parámetro de *maximum Pool Size*.
b. Para el caso de la aplicación reactiva, la conexión ha sido manejada por el archivo de configuración denominado *application.conf*, cuyo extracto del contenido se muestra a continuación:

```
# Database configuration
#  — — — —
# You can declare as many datasources as you want.
# By convention, t          he default datasource is named 'default'
#
db.default.driver=com.mysql.jdbc.Driver
db.default.url="jdbc:          mysql://192.168.52.100:3306/playconf          "
db.default.user=root
db.default.password=mysql
```

Se observan en las últimas líneas los parámetros de conexión que se requieren.

Para la aplicación "reactiva" no se ha variado ningún otro parámetro que pueda incidir en la performance del *pool* de conexiones. Para efectos del análisis se preparó un ambiente de pruebas (empleando un segmento de red no aislado, por lo que los resultados podrían incorporar "ruido") como se muestra en la figura 6, donde se aprecia la distribución del *software* dentro del entorno. Se debe mencionar que por consideraciones logísticas se han empleado equipos Intel i5, dado que esta es una primera aproximación al tema.

Figura 6. Entorno de prueba



Elaboración propia.

La versión tradicional (Java EE) se empaquetó en el formato estándar (.war) de la especificación Java EE y se desplegó vía consola de administración en un Servidor GlassFish 4.0 ejecutándose con Java JDK 1.7.0_67. El *driver* de conexión a la base de datos es el proporcionado por GlassFish 4. La versión reactiva se generó

a partir de la consola de TypesafeActivator y el servidor empleado viene embebido en esta (Netty server). El *driver* de conexión a la base de datos se obtuvo desde los repositorios de Maven por medio de los comandos de TypeSafeActivator.

Se empleó Apache JMeter 2.11 para generar las pruebas. Se definió un *script* de pruebas por cada aplicación para los URL: "/", "/new" y "/submit". Para el caso del URL "/submit" se tuvo que codificar la generación de datos simulados (mediante *scripts* de JMeter), los cuales se envían a la base de datos.

Previo al inicio de las pruebas, ambas aplicaciones fueron desplegadas en sus respectivos servidores.

El servidor GlassFish fue "afinado" teniendo en consideración:

a) En las opciones de Despliegue: se desactivó las opciones de "Auto-Deployment" y "Dynamic Application Reloading". Se activó la precompilación de JSP
b) En las opciones de Logging, se emplean los valores por defecto.
c) En las opciones de Web Container Settings: para las propiedades de manager y sesión se emplean los valores por defecto. Se desactiva la opción de "Dynamic JSP Reloading".
d) En parámetros de la JVM, se coloca -Xmx720m.
e) En parámetros de monitoreo se activa la opción para monitorear los *threads*.
f) Se modificaron los parámetros de "Network Listening Settings" y "Thread Pool Settings" del servidor Glassfish para soportar de 500 hilos de ejecución y una capacidad máxima de 500 conexiones en el *pool* JDBC.

Para el caso del servidor de Typesafe Activator se hicieron ajustes en la configuración de *threads*, como se aprecia en el extracto con las líneas que indica *parallelism-min* y *parallelism-max*, para generar tres escenarios de prueba:

```
akka {

    l oggers = ["akka.event.Logging$DefaultLogger",
    "akka.event.slf4j.slf4jLogger                    ]
        loglevel = WARNING
        db- dispatcher {
           #  Dispatcher is the name of the event                  - based dispatcher
           type = Dispatcher
           #  What kind of ExecutionService to use
           executor = "fork          - join   - executor"
           #  Configuration for the fork join pool
           fork   - join   - executor {
                #  Min number of threads           to cap factor        - based paralleli          sm
                number to      parallelism        - min = 10
                #  $ Max number of threads to cap factor                   - based
                parallelism number to              parallelism       - max   = 50
           }
    }
```

a) TypeSafe Activator *out of the box* (son los valores mostrados en el extracto).

b) TypeSafe Activator, con un máximo de 500 hilos paralelos.

c) TypeSafe Activator, con un máximo de 1000 hilos paralelos.

Para cada escenario planteado, la secuencia de prueba fue la siguiente:

1. Se limpió la base de datos (se ejecutó Drop y luego Create).
2. Se activó el servidor de aplicaciones.
3. Se ejecutó el *script* de pruebas desde la consola de JMeter.
4. Se recolectaron los resultados de la consola de JMeter

Esta secuencia de pasos se repitió para la simulación de 500, 1000, 1500 y 3000 usuarios.

5. Resultados

En esta primera aproximación, los resultados obtenidos en las pruebas se muestran en la tabla 3, la cual resume el valor total obtenido de la columna de Throughput en el "Summary Report" de JMeter. Se puede apreciar que la plataforma de TypeSafe Activator "out-of-the-box" es ampliamente superada por un GlassFish afinado específicamente para la prueba, situación que no ocurre si se deja al servidor GlassFish con los valores por defecto.

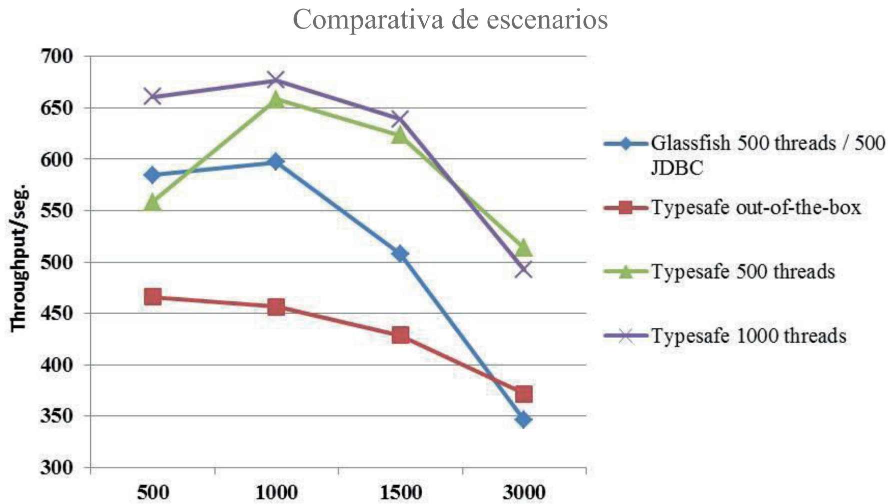
Tabla 3. Resultados

	500	1000	1500	3000
Glassfish 500 threads / 500 JDBC	584.5	597.0	507.7	346.1
Typesafe out-of-the-box	465.9	456.2	428.7	371.9
Typesafe 500 threads	558.4	658.4	623.5	513.2
Typesafe 1000 threads	660.7	676.9	639.3	492.4

Elaboración propia.

Por otro lado, se observa que al pasar de una concurrencia de 500 a 1000 usuarios simulados, la configuración de TypeSafe con 500 *threads* es la que presenta mejor escalamiento, como se puede apreciar en la figura 7, donde se nota la pendiente más pronunciada. Si se sigue incrementando la cantidad de usuarios simulados, el servidor GlassFish es el que degrada su *throughput* más rápidamente.

Figura 7. Resultados de las pruebas



Elaboración propia.

Aunque por decisiones logísticas y académicas no se han podido realizar las pruebas, se infiere que cualquiera de los servidores de aplicaciones JEE más utilizados (por ejemplo IBM WebSphere, Oracle WebLogic o incluso JBoss) presentaría el mismo comportamiento, dada la estructura interna muy similar entre todos ellos.

6. Conclusiones

Este artículo presenta una primera exploración al tema de aplicaciones reactivas y propone mejorar la escalabilidad de aplicaciones web empleando este nuevo paradigma de programación, donde una primera experiencia demuestra que la conjunción de una aplicación ejecutándose en servidores especialmente diseñados para soportar las características de reactividad genera una ganancia importante en el rendimiento ante situaciones de concurrencia.

Es importante analizar la escalabilidad desde el punto de vista del *software* si es que no se cuenta con presupuesto para renovar o ampliar la infraestructura de servidores físicos. A la vez, una limitación importante es la curva de aprendizaje de este nuevo paradigma de programación, sobre todo en proyectos considerados críticos para una empresa.

Un tema pendiente de evaluar son las decisiones financieras respecto a costos de licenciamiento y soporte de servidores JEE tradicionales comparados

con plataformas reactivas como las descritas en el artículo. Sin embargo, podrían aplicarse rápidamente los resultados de este artículo en la implementación de aplicaciones de uso concurrente como el registro de aportantes para la devolución del Fonavi, la consulta en la ONPE para verificar a los miembros de mesa, la consulta del RUC en la Sunat, etcétera.

Se propone que un futuro trabajo se enfoque en aspectos de monitoreo de las aplicaciones reactivas.

Referencias

Broadwell, P. (2004). Response time as a performability metric for online services. *Computer Science Division University of California at Berkeley*. Recuperado de http://digitalassets.lib.berkeley.edu/techreports/ucb/text/CSD-04-1324.pdf

Brown, K., & Capern, M. (2014). *Top 9 rules for cloud applications*. IBM Developer Works. Recuperado de http://www.ibm.com/developerworks/websphere/techjournal/1404_brown/1404_brown.html

Charbonneau, P.-H. (2011). java.net.socketinputstream.socketread0 hangs thread. *Java EE Support Patterns*. Recuperado de http://javaeesupportpatterns.blogspot.de/2011/04/javanetsocketinputstreamsocketread0.html

IBM Knowledge Center (2011). *WebSphere Application Server Network Deployment 8.0.0*. Recuperado de http://www-01.ibm.com/support/knowledgecenter/SSAW57_8.0.0/com.ibm.websphere.installation.nd.doc/info/ae/ae/rprf_queuetip.html?cp=SSAW57_8.0.0%2F1-5-0-4-3-1&lang=es

Kambhampaty, S., & Srinivas, V. (2007). *Performance modeling for web based J2EE and .NET Applications. International journal of computer, information, systems and control engineering, 8*(1), 2567-2573.

Kegel, D. (1999). The C10K problema. Dan Kegel's Web Hostel. Recuperado de http://www.kegel.com/c10k.html/

Kounev, S., & Buchmann, A. (2003a). *Performance modelling and evaluation of large-scale j2ee applications*. International Conference of the Computer Measurement Group (CMG) on Resource Management and Performance Evaluation of Enterprise Computing Systems - CMG2003. Recuperado de https://sdqweb.ipd.kit.edu/publications/descartes-pdfs/KoBu2003-CMG-PerfModeling_J2EEApps.pdf

Kounev, S., & Buchmann, A. (2003b). *Performance modelling of database contention using queueing petri nets*. IEEE International Symposium on Performance

Analysis of Systems and Software (ISPASS'03). Recuperado de https://www.dvs.tu-darmstadt.de/publications/pdf/03-ispass-QPNs.pdf

Kounev, S. (2005). *SPECjAppServer2004: The New Way to Evaluate J2EE Performance*. Recuperado de http://www.oracle.com/technetwork/articles/entarch/specjappserver2004-083982.html?ssSourceSiteId=otnes.

Layka, V. (2014). *Learn java for web development: Modern java web development*. Recuperado de http://it-ebooks.info/book/3845/

Liotopoulos, F. (2005). *Performance modelling and analysis of a large-scale e-Voting systems*. International Conference on Performance Modelling and Evaluation of Heterogeneous, pp. 1-9.

Menascé, D., Barbará, D., & Dodge, R. (2001). *Preserving QoS of e-commerce sites throrugh self-tunning: a performance model approach. ACM conference on electronic commerce*. Recuperado de http://cs.gmu.edu/~menasce/papers/menasce-acmec01.pdf

Microsoft Developer Network (2014). *Escalabilidad*. Recuperado de http://msdn.microsoft.com/es-es/library/aa292172(v=vs.71).aspx

Real Academia Española. (2014). *Diccionario de la lengua española* (23.ª edición). Recuperado de http://www.rae.es

Taing, N. (2011). *Thread Pool. Lycog programming, concurrency, distributed systems, simulation*. Recuperado de http://lycog.com/distributed-systems/thread-pool

Typesafe. (s. f.). *Play framework for java developers. Online training*. Recuperado de https://typesafe.com/how/online-training/play-java