Comparación de desempeño entre los estilos arquitecturales REST y Actores

Pedro A. Otoya Visbal

Asesorado por: PhD. Kelly Garcés Pernett

Tesis de grado para optar por el título de
Ingeniero de Sistemas y Computación

Universidad de Los Andes
Bogota, Colombia
Mayo 2016

Abstract

La intención del proyecto es comparar dos estilos arquitectónicos que se pueden aplicar en el desarrollo de aplicaciones web transaccionales. Para lograr esto, se parte de una aplicación REST implementada en la plataforma J2EE con JAX-RS, y luego esta se migra a la plataforma Play Java, que sigue el modelo de actores. Acto seguido, se realizan distintos despliegues en la nube, pruebas de carga y se aplican tácticas para mejorar el desempeño de la arquitectura Play Java. A partir de los resultados, se desarrolla una caracterización de las tácticas más adecuadas para satisfacer ciertos requerimientos.

Reconocimientos

Agradezco a la profesora Kelly Garces Pernett por su importante contribución a mi formación, y su apoyo en este proyecto. Por último, quiero expresar mi gratitud hacia mis padres, por a su persistencia y respaldo en mis estudios profesionales.

Tabla de Contenido

Tabla de figuras

1 Introducción

Los servicios de infraestructura en la nube han marcado una fuerte tendencia en las arquitecturas web. La premisa principal de la nube consiste en suplir la demanda de carga, utilizando varias máquinas de menor tamaño, en vez de aumentar los recursos computacionales de una misma máquina. Para soportar lo anterior, se usan arquitecturas reactivas basadas en el estilo de actores.

La plataforma Play es hoy en día una de las pioneras en el estilo de actores. Implementada sobre el modelo de actores Akka, permite construir aplicaciones web que sean livianas, sin estado, no bloqueantes, y con alta escalabilidad. En pocas palabras, facilita la construcción de sistemas que tengan buen desempeño en máquinas con recursos limitados.

En contraparte está el estilo REST, el cual compete a la gran mayoría de aplicaciones implementadas en el mercado. Dentro de este grupo, hay una gran cantidad de sistemas escritos en Java que usan JAXRS soportados en la popular plataforma *Java Enterprise Edition.*

Los estilos mencionados son similares en la estructura y responsabilidad de los elementos de alto nivel. Sin embargo, en el bajo nivel hay una gran diferencia en los modelos de concurrencia que utilizan. En el caso de la plataforma Play, las peticiones se procesan de manera asíncrona en el servidor, lo cual permite que los hilos *no sean bloqueados* durante el ciclo de vida de cada una de estas. Lo opuesto ocurre con los sistemas que utilizan JAXRS, ya que en estos cada hilo está atado al ciclo de vida de la petición, y por lo tanto este queda *bloqueado*.

En vista a lo anterior, se pretende proveer una solución arquitectónica que permita mejorar el desempeño de una aplicación web transaccional teniendo en cuenta la optimización en el uso de recursos computacionales.

En este proyecto se comparan los dos estilos mencionados, instanciados en dos despliegues distintos. Ambos siguen un patrón monolítico, sin embargo uno se implementa en Play Java y el otro en JEE usando JAXRS.

2 Estado del Arte

Los estilos arquitecturales REST y Actores, imponen restricciones a los sistemas, y se conoce que ambas ofrecen buen desempeño. Los dos patrones de diseño que más implementan el estilo REST, son las monolíticas y de micro servicios. En el caso de las arquitecturas reactivas, generalmente se usa en un patrón monolítico, puesto que el compromiso del modelo de actores, es garantizar la escalabilidad sin incurrir en altos tiempos de desarrollo. Dicho esto, ha habido un conjunto de escritos que han trabajado en comparar estas arquitecturas, en términos de atributos de calidad varios.

Aparicio *et al* en (Aparicio, 2015), realizo una comparación entre las arquitecturas de micro servicios y reactivas. Esto lo hizo con el objetivo de revisar cuál de las dos, se comportaba mejor en términos de tolerancia a fallos, y elasticidad. Para esto, implemento una aplicación web siguiendo el patrón de micro servicios típicos, el cual consiste en un balanceador de carga que redirección peticiones hacia los distintos nodos donde se ubican los micros servicios. Además de esto, desplego otra aplicación web equivalente, desarrollada en Play, pero usando una arquitectura de clúster reactiva, basada en la implementación de actores Akka. Luego de realizar pruebas, llego a la conclusión de que los micro servicios eran más tolerantes a la inyección de fallas, pero que la arquitectura reactiva tenía mayor elasticidad.

Por otro lado, Villamizar *et a.* en (Villamizar, 2015), llevaron a cabo un proyecto que comparo una aplicación web siguiendo una arquitectura monolítica contra micro servicios. Ambas aplicaciones fueron implementadas en Play y desplegadas en la plataforma de nube AWS. En este escrito desarrollaron pruebas para dos tipos de servicios, uno con procesamiento intensivo, y otro con procesamiento liviano. Los resultados de las pruebas, indicaron que en el servicio de procesamiento intensivo, micro servicios se comportaba mejor, mientras que en el servicio de procesamiento liviano, la arquitectura monolítica tenía menores tiempos de respuesta.

Los trabajos pasados sobre comparación de arquitecturas, no han enfatizado en la comparación que este proyecto pretende. Generalmente, se debe a que dan por hecho que los patrones implementados en Play, siempre van a ser superiores en cuanto a desempeño. Sin embargo, Foy *et al* en (Foy, 2014), menciona las aplicaciónes transaccionales propensas a tener peticiones demoradas, deben recibir tratamiento especial, en haras a aprovechar las ventajas que ofrece la concurrencia de los actores. Un ejemplo son las peticiones dependientes de consultas a bases de datos externas, que al realizarse de manera atomica, bloquean la ejecucion y deben recibir un tratamiento especial. Según lo anterior, la motivacion de este proyecto consiste en instanciar tacticas (Bass, 2013) que garanticen un optimo desempeño del modelo de actores, en aplicaciones web con alta naturaleza transaccional.

3 Objetivos

3.1 Objetivo general

Comparar aplicaciones web hechas en REST y Actores, en términos de desempeño, para distintos casos de prueba

3.2 Objetivos específicos

- Migrar una aplicación REST existente a Play Java.
- Definir e implementar pruebas que permitan verificar métricas de desempeño de las aplicaciones.
- Analizar e interpretar las métricas resultantes.
- Modificar la arquitectura para mejorar las métricas.
- Concluir cuáles fueron las tácticas más apropiadas para mejorar las métricas.

4 Fundamentación

El modelo de actores fue inventado en los años 70. Sin embargo, hoy en día es la clave para la implementación de arquitecturas reactivas, debido a su facilidad para manejar la concurrencia. Siguiendo ese contexto aparece *Akka*, una compañía que implementó el modelo de actores en el lenguaje Scala y desarrollo las interfaces para que se pueda utilizar en Java. La relevancia de *Akka actors* se debe a que, esta es el Core sobre el cual esta implementado la plataforma reactiva Play.

4.1 Akka Actors

Akka actors, es una plataforma que implementa el modelo de actores (Akka, s.f.). Su objetivo consiste en proveer una herramienta que permita aislar al programador de las dificultades que surgen al implementar concurrencia con los métodos tradicionales. Es decir, involucrar semáforos, señalamiento, sincronización, candados, etc. Este modelo es completamente asíncrono y consiste en partir tareas grandes y delegar los pedazos restantes a los distintos actores. Cada actor encapsula dos cosas, estado y comportamiento. El estado hace referencia a las variables privadas que tiene el actor, mientras que el comportamiento es lo que ejecuta el actor una vez lee un mensaje. Es importante tener en cuenta que los mensajes, son la única interface entre los actores y cualquier otro componente. Por lo tanto, el diseño de los mensajes debe ser hecho con cautela.

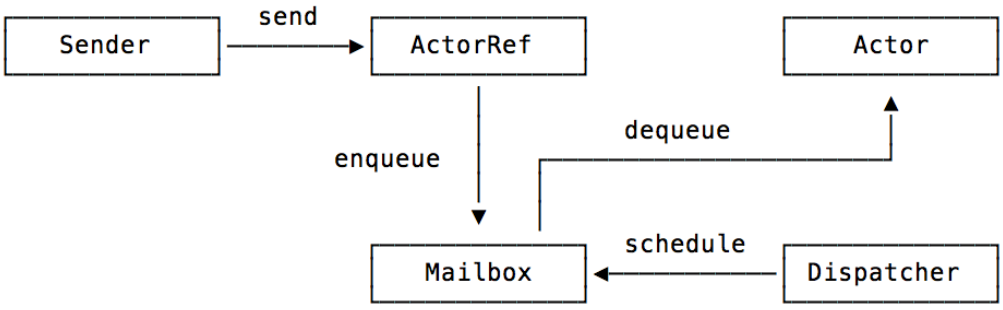

Figura 1. Diagrama de contexto del modelo de actores. Este describe las acciones que toma cada componente para procesar el flujo mensajes.

Los significados y tareas de cada componente básico mostrado en la figura son los siguientes:

- *ActorRef* representa la referencia a un actor. Su responsabilidad consiste en apuntar hacia un conjunto de actores, teniendo así la carga de manejar el ciclo de vida de estos – pues tiene la posibilidad de eliminar actores solo quitándoles la referencia, en cuyo caso el recolector de basura los borraría de la memoria.

- *Actor* es la clase que representa el actor. Tiene la responsabilidad de guardar variables privadas y define la lógica para procesar los mensajes que desencola en su mail box.
- *Mailbox* es una cola que tiene cada actor, destinada a recibir los mensajes que deben ser procesados. El tamaño de esta cola es fijo y se puede parametrizar.
- *Dispatcher* es el componente que decide cuando desencolar el siguiente mensaje, y luego asignarle un hilo de ejecución al actor para que este pueda procesarlo. Este elemento es clave para implementar tácticas de desempeño.

4.1.1 Despachador de mensajes en Akka

El despachador de Akka es el componente más importante en la ejecución de los actores y el procesamiento de sus mensajes. Básicamente, el despachador está respaldado por un ejecutor. Estos dos, funcionan en un patrón productor-consumidor, en donde el despachador "produce" tareas, y el ejecutor las "consume". En Akka, los despachadores ejecutan en un grupo de hilos dedicados para ellos. Tal como se muestra en la figura 2, este se encarga de elegir un actor con su mensaje, y le asigna un hilo para que pueda ejecutarse. Ahora, el despachador por defecto de Akka, es basado en eventos, lo cual significa que los mensajes son procesados en orden cronológico.

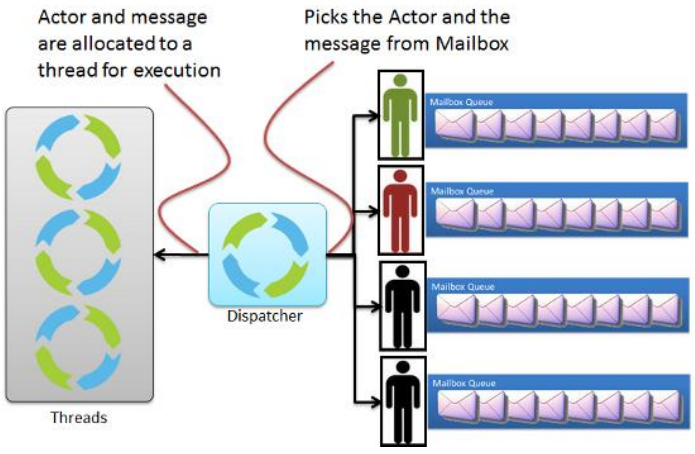

Figura 2. Diagrama de contexto que ilustra la implementación del despachador en Akka (Gupta, 2012).

4.1.2 Executor Framework de Java

El *Executor Framework* de Java, implementa ejecutores que son utilizados por Akka para la ejecución de tareas asíncronas. Cada ejecutor se inicializa con un grupo de hilos y una cola de tareas ejecutables. Cada vez que el despachador de Akka programa la ejecución de un actor, este se encola en la cola de tareas del ejecutor. Luego pasa a ser desencolada y procesada por el primer hilo libre en el grupo.

Sumado al anterior, está el hecho de que hay dos tipos de ejecutores distintos. El primero de ellos es el *ThreadPoolExecutor*, que consiste en un grupo de hilos cuya labor es consumir tareas provenientes de la cola de tareas. Por otro lado, el *ForkJoinPoolExecutor* tiene una cola de tareas y un grupo de hilos igual que el anterior, pero con la salvedad de que cada *worker* tiene su propia cola. En la figura 3 se pueden ver todas las acciones que realiza este ejecutor. Se muestra una ejecución con 2 *workers*, por lo tanto se ven 2 colas locales y una cola global. La cola global recibe las tareas a ejecutar, que luego son desencoladas por A y B. Cuando estos workers desencolan una tarea, realizan un algoritmo recursivo para partirla en pedazos más pequeños, que luego son almacenados en sus colas locales haciendo *push.* Una vez empiezan a procesar, realizan un *pop* para extraer la siguiente tarea que está en su cola local. Nótese que en la cola local de A, las tareas más grandes quedan en la parte trasera de la cola, por lo tanto son las ultimas en ejecutar. Por último, se muestra una situación en la que B, tiene su cola vacía, y por lo tanto realiza un *steal* sobre la cola de A, con el objetivo de ayudarle a terminar sus tareas. Esto último, se conoce como *work-stealing*.

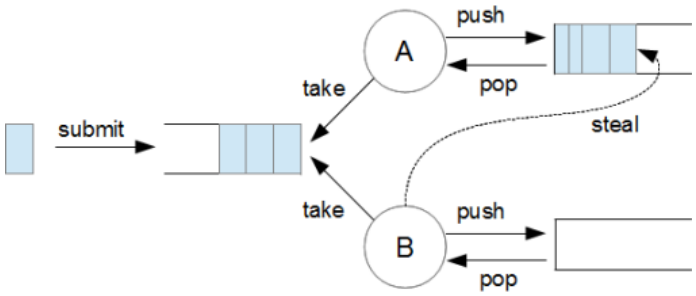

Figura 3. Ejemplo de un ForkJoinPool con 2 workers. (Peschlow, 2012)

Adicionalmente, es importante mencionar que el desempeño de estos ejecutores está ligado al tipo de tareas que ejecutan. Por ejemplo, el *ThreadPoolExecutor* funciona mejor con tareas atómicas, ya que hay un hilo comprometido a cumplir esa tarea lo más rápido posible. En contraste, si la tarea no es atómica, el funcionamiento del *ForkJoinPoolExecutor* es óptimo, ya que este puede romper la tarea en pedazos pequeños, y así beneficiarse del paralelismo. Estos conceptos serán vitales para justificar la selección de ejecutores realizada en la sección 5.2.2.

4.2 Play Framework para Java

Play es una plataforma que permite desarrollar aplicaciones web con una arquitectura de componentes livianos y sin estados. Esta, se puede programar en Java o Scala y ejecuta sobre la JVM. Además, está construida encima de Akka, y por lo tanto utiliza el modelo de actores para proveer una alta escalabilidad, utilizando recursos mínimos. Esta última característica resalta el atractivo de Play en la nube, pues puede brindar un desempeño excelente, en máquinas de pequeños tamaños, incurriendo así en una reducción de costos a pagar por la infraestructura.

Llegado a este punto, surge la importancia de responder a la pregunta: ¿Cómo es la estructura interna de Play, y de qué manera utiliza Akka? Para responder a esto, se deben introducir ciertos elementos lógicos de la arquitectura correspondiente a la plataforma. En primer lugar, Play funciona en una arquitectura cliente-servidor, y sus componentes se organizan siguiendo un patrón MVC. La figura 4, muestra una descripción general de la arquitectura del sistema.

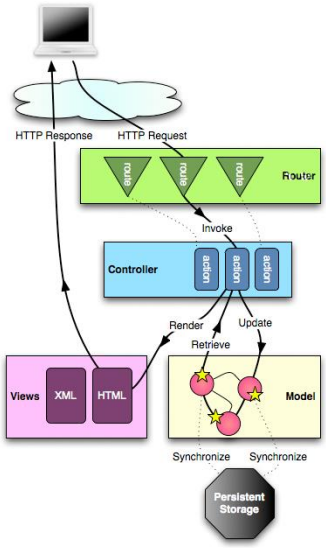

Figura 4. Diagrama de componentes ilustrando la arquitectura de Play. Nótese que Play sigue un patrón MVC en el lado del servidor.

El ciclo de vida de una petición HTTP comienza en el *Router*, el cual se encarga de recibirla y re direccionarla a una acción implementada en cierto controlador especifico. Cada controlador implementa un conjunto de acciones, y estas se encuentran asociadas a una petición HTTP específica. Las acciones se encargan de procesar sus peticiones, invocando la lógica de negocio que se encuentra en el modelo, y luego produciendo una respuesta a través de la vista. Ahora, estas peticiones HTTP tienen una característica especial, y es que son reactivas. Esto implica, que el servidor maneja cada petición y respuesta de manera asíncrona y no bloqueante. Dicha característica, es la clave del alto rendimiento que ofrece Play.

Para cumplir con las características asíncronas y no bloqueo, Play se soporta en Akka y Java NIO. Akka tiene la responsabilidad de usar actores para el procesamiento asíncrono de cada petición y respuesta, mientras que Java NIO implementado por Netty, provee las funcionalidades del servidor web. La figura 5 muestra la ejecución interna de un HTTP GET.

Particularmente, es indispensable tener en cuenta que cada invocación a los actores, se realiza enviando un mensaje hacia su mail box, y por lo tanto la ejecución es asíncrona. Dichos actores, son creados por cada controlador, y la interacción entre ellos varía dependiendo del tipo de petición. En base a la figura 5, el actor consumidor, recibe la petición y la re direc-

ciona hacia un actor productor, quien realiza la invocación al servidor web donde se ejecutan las acciones del controlador. Una vez la acción termina el productor la recibe y la envía al actor transformador quien se encarga de post procesar la petición, y re direccionarla de nuevo al actor consumidor, el cual envía la respuesta Http al cliente web.

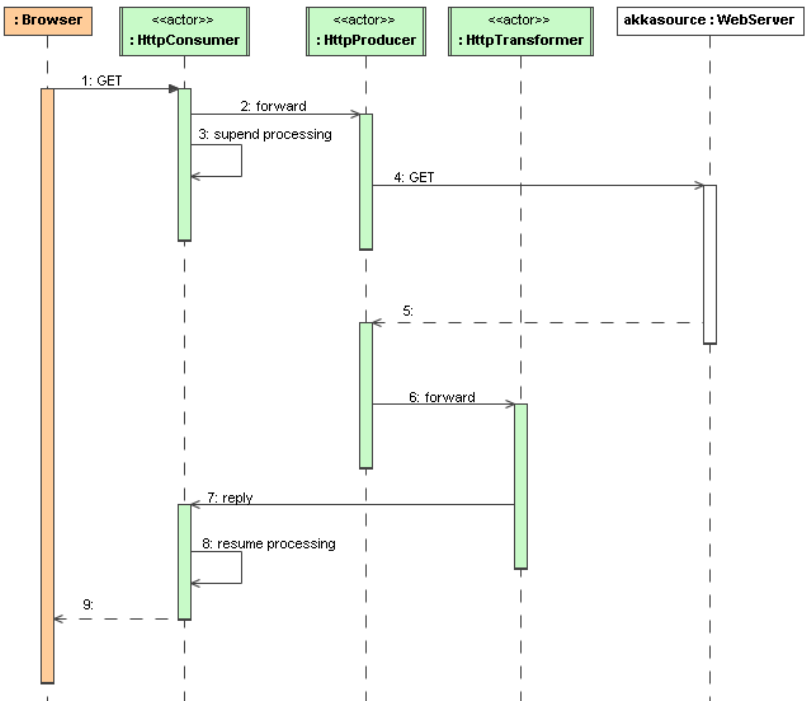

Figura 5. Diagrama de secuencia ilustrando el manejo interno de las peticiones utilizando actores de Akka y el servidor web Netty con Java NIO. (Akka, s.f.)

4.2.1 Modelo de concurrencia

Anteriormente, se realizó un acercamiento al funcionamiento global del servidor de Play. Sin embargo, el modelo de concurrencia es la última pieza esencial para comprender la manera en la cual se ejecutan ciertas funcionalidades de la aplicación. La característica principal del modelo es que las acciones ejecutadas en los controladores, son no bloqueantes, lo cual implica que su ejecución completa puede ser realizada por distintos hilos. Esto no sucede en los servidores comunes, como por ejemplo los *Servlets* de Java, donde cada petición se ejecuta en un mismo hilo de manera bloqueante. Estos dos modelos se conocen como el servidor concurrente y servidor por eventos.

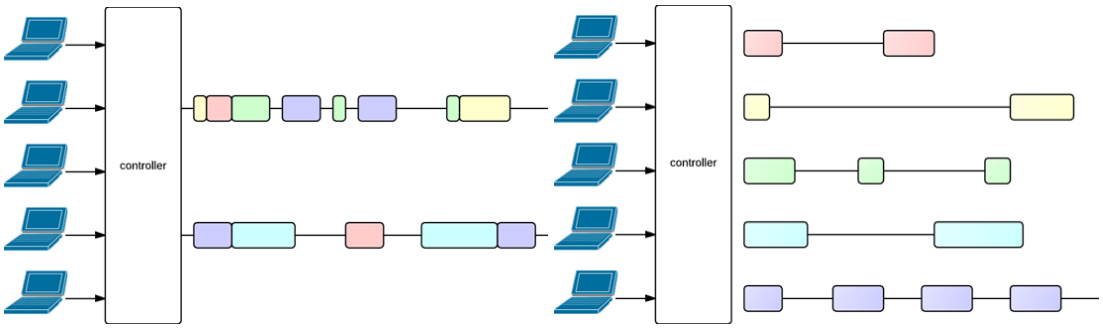

Figura 6. Comparación entre el modelo de ejecución concurrente (izquierda) y el modelo de ejecución basado en eventos (derecha) (Foy, 2014)

En la figura 6 se puede apreciar las ventajas que tiene el modelo de Play (i.e. basado en eventos), contra el modelo concurrente que se usa en la mayoría de los servidores web tradicionales. Principalmente, es destacable el hecho de que ambos tardan el mismo tiempo ejecutando las tareas de 5 clientes, pero el modelo basado en eventos lo hace utilizando 2 hilos, mientras que el otro utiliza 5 hilos. Esto último, se debe a que el no bloqueo permite que cada hilo quede libre para procesar múltiples acciones, permitiendo que cada hilo este trabajando la mayor cantidad de tiempo. En pocas palabras, permite un uso más eficiente de los hilos creados.

5 Metodología de solución

Con el fin de realizar comparaciones entre los dos estilos arquitecturales, se partió de un proyecto existente desarrollado en JEE con JAXRS, que luego fue migrado al Play Framework. El proceso de migración se realizó bajo la metodología ACDM. Esto último, significa que la arquitectura se perfecciono de manera iterativa, con el fin de que la aplicación tuviera el mejor desempeño posible en una maquina especifica. Para los despliegues, se utilizaron máquinas de la PaaS Heroku, junto con una base de datos PostgreSQL alojada en una t2.micro de AWS.

5.1 Caso de estudio

El caso de estudio usado para validar las arquitecturas, corresponde a una aplicación web transaccional llamada MarketPlace. Esta fue creada por el generador *CRUD Maker*, a partir del modelo relacional presentado en la figura 7. Dicho generador, utiliza produce una aplicación web tradicional en JEE, con una arquitectura en 4 capas: Web-Servicios-Lógica-Persistencia. Sin embargo, para medir el desempeño se usan únicamente las últimas 3 capas. La capa de servicios implementa el estilo REST utilizando JAXRS. Por otro lado, la lógica utiliza EJB Stateless, y la persistencia es basada en JPA con el proveedor EclipseLink.

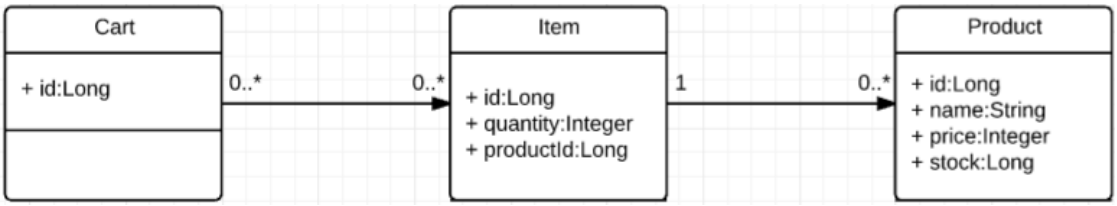

Figura 7. Modelo relacional del proyecto MarketPlace.

Basado en el caso de estudio se definieron 4 casos de uso que debían implementarse. Estos se describen a continuación:

1. **Crear un nuevo producto**
   Esta funcionalidad corresponde a agregar un nuevo registro en la tabla *Producto*.
2. **Crear un nuevo carrito**
   En este caso de uso se agrega un nuevo carrito en el sistema. Aquí cabe recalcar que deben agregarse un nuevo registro en la tabla *Cart*, y cada uno de sus ítems debe ser agregado en la tabla *Item*. Además, por cada ítem se debe actualizar el stock del *Producto* correspondiente.

3. **Obtener productos más vendidos**

   Este es un caso de procesamiento intensivo. Por esta razón se ejecuta un algoritmo en el servidor, que se encarga de revisar los productos más vendidos, a partir de todos los ítems. Normalmente esto es una consulta a la base de datos, sin embargo para hacerlo intensivo en CPU, se realizó la implementación del lado del servidor.

4. **Obtener todos los carritos**

   En este caso se obtienen todos los carritos existentes en la base de datos.

5.1.1 Despliegue del monolítico Marketplace en JEE

La aplicación base del Marketplace en JEE (i.e. la generada por el CRUDMaker), tuvo que ser adaptada al modelo de despliegue que provee Heroku en sus máquinas. Para esto fue necesario utilizar el Framework Apache TomEE, el cual permitió embeber el servidor web junto con su contenedor de aplicaciones, en un mismo proyecto ejecutable. De esta manera, fue posible desplegar la aplicación empresarial en un *HerokuDyno*. Los detalles del despliegue se pueden ver en la figura 8.

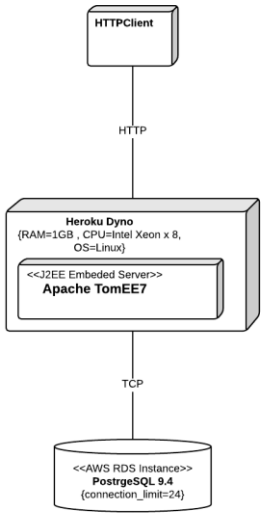

Figura 8. Despliegue de la arquitectura JEE monolítica.

5.1.2 Afinamiento del servidor empresarial TomEE7

Para el afinamiento de la aplicación Marketplace JEE, se realizaron modificaciones sobre los siguientes grupos de hilos:

- Hilos de conexiones JDBC
- Hilos del receptor Tomcat

En esta etapa, los grupos de hilos se fijaron en el máximo número de conexiones permitido por la base de datos, es decir 24 hilos. El diagrama de concurrencia en la figura 8a, muestra un *ThreadPool* encargado de procesar las solicitudes, y un *JdbcConnectionPool* usado para comunicarse con la base de datos.

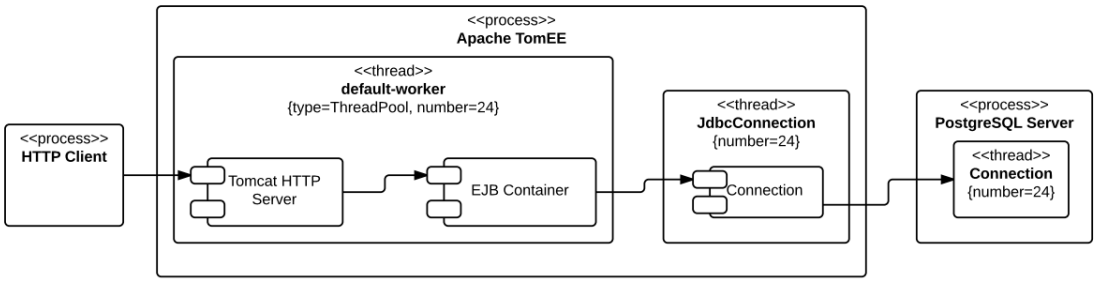

Figura 8a. Diagrama de concurrencia correspondiente a la arquitectura JEE Monolitica.

5.2 Migración del Marketplace hacia el Play Framework para Java

Durante la migración del Marketplace a Play, se realizaron las implementaciones de los 4 casos de uso, teniendo en cuenta detalles específicos para obtener buen desempeño. Para lograr esto se implementaron los modelos, siguiendo la figura 7, y luego se crearon acciones en los controladores responsables de la funcionalidad. La figura 9, muestra la vista de desarrollo, en la cual se puede ver cómo está organizado el código fuente de la aplicación. Por otro lado, el *Router* fue parametrizado en el archivo *conf/routes*, quien contiene las relaciones entre las peticiones HTTP y las acciones que ejecuta cada uno. En la figura 10, se muestra el archivo *routes* del Marketplace en Play.

En el caso del despliegue, se replicó el ambiente usado para el Marketplace JEE. Esto significa que se usó la misma base de datos, y un HerokuDyno con la misma capacidad computacional. Todo esto con la finalidad de que los resultados, obtenidos en las pruebas, fueran comparables.

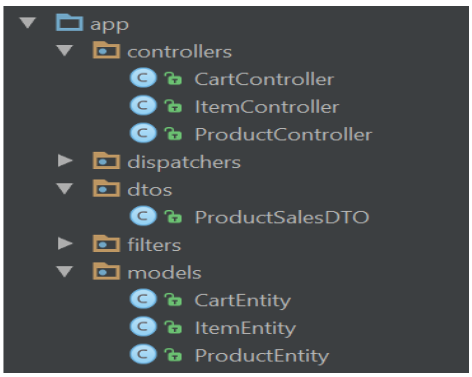

Figura 9. Vista de desarrollo correspondiente a la aplicación Play.

```
# CartController
GET   /carts                          controllers.CartController.getCarts
POST  /carts                          controllers.CartController.createCart

# ItemController
GET  /items/mostselledproducts        controllers.ItemController.getMostSelledProducts

# ProductController
GET  /products                        controllers.ProductController.getProducts
POST  /products                       controllers.ProductController.createProduct
```

Figura 10. Archivo *conf/routes* de la aplicación Play

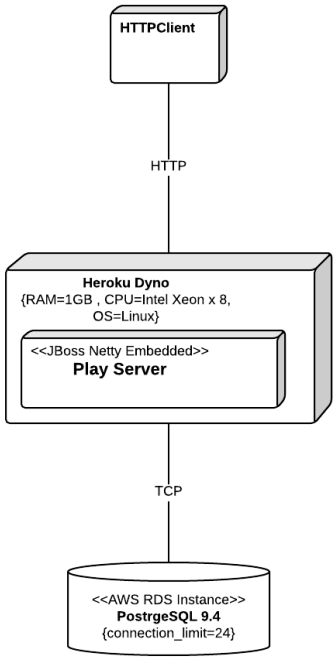

Figura 11. Despliegue monolitico del Marketplace en Play.

5.2.1 Detalles de implementacion

El no bloqueo, fue la premisa principal que se tuvo en cuenta para la implementación. Ahora, debido a que todas las funcionalidades de la aplicación requerían de llamadas a la base de datos, fue pertinente investigar las librerías de drivers disponibles. En este caso, se encontró que el driver JDBC era el más usado y confiable. Sin embrago, en su implementación bloquea el hilo mientras consulta la base de datos. Por esta razón, surgió la necesidad de utilizar las nuevas librearías, asincrónicas y funcionales, que brinda Java 8.

En Java 8, se encuentra el Framework de promesas. Las promesas en Java se conocen como *CompletableFuture*, y son objetos que permiten ejecutar bloques de código asincrónicamente. Este concepto se puede entender inspeccionando el código de la figura 12. Aquí, el controlador implementa una acción asíncrona, ya que no retorna un resultado sino la promesa de un *Result* Http. En pocas palabras, esto significa que en memoria se registra la ac-

ción en una cola de eventos, y luego esta, suspende su ejecución hasta que el futuro termine de procesar la petición. Para lograr esto, se utiliza el método *CompletableFuture.supplyAsync*, quien se encarga de ejecutar la consulta base de datos en una tarea asíncrona. Una vez terminada la consulta, el futuro invoca el método *thenApply*, encargado de ejecutar el *callback* que retorna el *Result* Http. ***Es importante notar que los futuros pueden ejecutarse en contextos de ejecución dedicados, que son diferentes al cual los inicializo***. Dicho tema será abordado con mayor detalle, en la sección 5.2.2.

```
/**
 * Accion que procesa la solicitud de obtener carritos
 * @return La promesa de un Resultado HTTP (i.e. ok 202, not found 404, etc)
 */
public CompletionStage<Result> getCarts(){
    MessageDispatcher jdbcDispatcher = AppExecutionContexts.jdbcDispatcher ;
    return CompletableFuture.supplyAsync(()->{
        return CartEntity.finder.all();
    }, jdbcDispatcher).thenApply(cartEntities -> {
        return ok( Json.toJson( cartEntities ) );
    });

}
```

Figura 12. Implementacion asincrona de la accion encargada de obtener carritos.

5.2.2 Contextos de ejecución

Un contexto de ejecución se refiere a la pareja despachador-ejecutor. En Play los contextos de ejecución son las construcciones esenciales que permiten implementar las tácticas de paralelismo y concurrencia. Por tal motivo, usarlos adecuadamente, y con la parametrización optima, permite obtener el desempeño ideal de la aplicación.

Los despachadores y ejecutores fueron introducidos anteriormente. Sin embargo, es vital aterrizar su uso en la aplicación que corresponde al caso de estudio. Un contexto de ejecución se define en el archivo *conf/application.conf*, tal cual como se muestra en la figura 13. En esta se ven 3 contextos de ejecución distintos. El primero es el contexto por defecto, quien es responsable de ejecutar el modelo de actores dedicado a procesa las solicitudes Http – o sea los manejos de las peticiones, la comunicación con Netty, la invocación de acciones usando el Router, etc. Los otros dos contextos fueron definidos para ejecutar tareas dedicadas. Esto lleva a formularse una pregunta: ¿Por qué es necesario usar contextos de ejecución especiales?

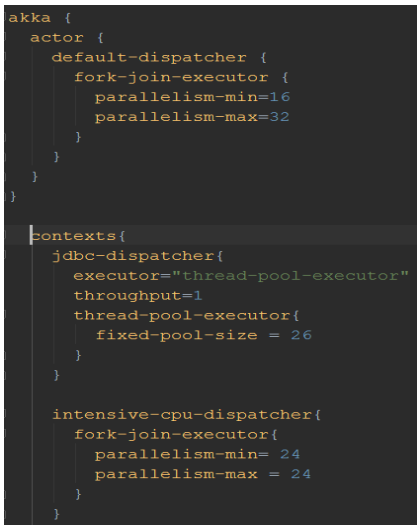

Figura 13. Creación de los contextos en el archivo *conf/application.conf*.

Los contextos de ejecución se pueden ver como un grupo de hilos, usados por la aplicación para ejecutar bloques de código. Siendo esto así, al usar múltiples contextos se está aumentando el número de hilos globales que la JVM tiene en ejecución. La utilidad de esto es que permite correr ciertas funcionalidades de negocio en contextos dedicados, que luego se parametrizan en base a estas. Foy *et al* en (Foy, 2014), explica que los drivers JDBC son bloqueantes-pues retienen el hilo esperando respuesta de la base de datos. Por esta razón, Foy recomienda que se utilice un contexto de ejecución dedicado para las consultas a base de datos, pues esta medida impediría que el contexto por defecto sea bloqueado, permitiendo así que sus hilos queden libres para trabajar en el procesamiento de las peticiones/respuestas HTTP. Este efecto se puede ver con claridad en el diagrama de concurrencia presentado en la figura 14. Los contextos ilustrados en el diagrama son los siguientes:

- ***default-context*** es el contexto por defecto sobre el cual funciona la aplicación Play. En este se ejecuta todo el manejo de las peticiones, y por lo tanto es quien corre el código implementado en los controladores. El ejecutor elegido es un ForkJoin, ya que los controladores se implementaron de manera asíncrona y por ende no bloquean.
- ***db-access-context*** es un contexto dedicado para las consultas a la base de datos. En este caso se eligió un ejecutor de tipo ThreadPool, ya que estas operaciones son totalmente atómicas.
- ***cpu-intensive-context*** es un contexto dedicado a tareas que sean intensivas en CPU. Aquí se eligió un ejecutor de tipo ForkJoin, ya que este tipo de tareas no son atómicas y pueden beneficiarse del paralelismo.

Cabe recalcar, que los números de hilos para cada contexto, fueron asignados realizando un afinamiento - el cual será detallado a fondo en la próxima sección.

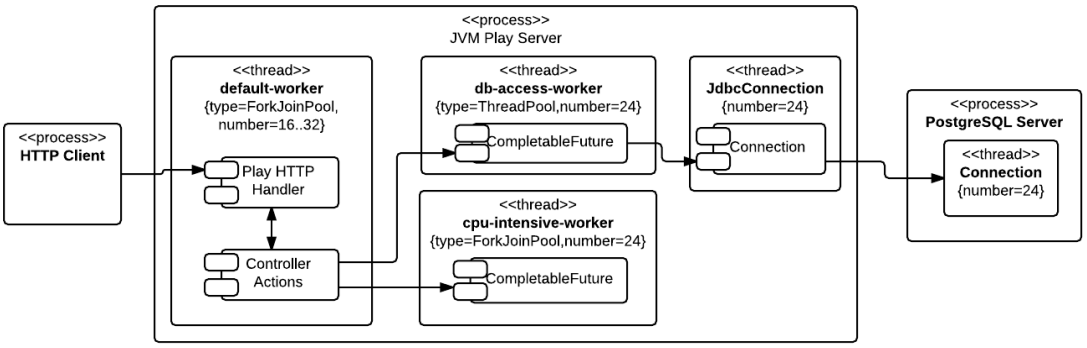

Figura 14. Diagrama de concurrencia correspondiente a la aplicación Play.

5.2.3 Implementación y afinamiento de los contextos

La implementación y afinamiento de los contextos marcaron dos etapas iterativas y dependientes del desempeño resultante. El proceso iniciaba en la etapa de implementación, en la cual se realizó una caracterización de las funcionalidades. Posteriormente, se realizó el afinamiento del número de hilos para los contextos implementados.

El objetivo de la primera fase, consistió en descomponer cada funcionalidad en sus operaciones más básicas, y de ahí decidir a cual se le podía asignar un contexto de ejecución dedicado. Para esto los criterios principales fueron: (1) evitar bloqueos, y (2) paralelizar las operaciones costosas que no fueran atómicas. El detalle de la caracterización se puede ver en las tablas correspondientes a las figuras 15 y 16. En la figura 15 se realiza un mapeo entre los servicios y las operaciones que estos realizan. Por otro lado, la figura 16 relaciona ejecutores, con cada una de estas operaciones. Utilizando estas tablas es posible determinar cuáles son los contextos que participan en la ejecución de cada servicio. Nuevamente, cabe recalcar que la selección se hizo de manera iterativa – se implementaban cambios que luego se verificaban por medio de pruebas y monitoreo.

	ObjectToJson	Read BD	Computo Intensivo	JsonToObject	Write BD	HTTP Handle
Obtener todos los carritos	X	X				X
Obtener productos mas vendidos	X	X	X			X
Crear un nuevo producto				X	X	X
Crear un nuevo carrito		X		X	X	X

Figura 15. Tabla mostrando las operaciones de cada funcionalidad.

	db-access-context(ThreadPool)	cpu-intensive-context(ForkJoin)	default-context(ForkJoin)
ObjectToJson			x
Read BD	x		
Computo Intensivo		x	
JsonToObject			x
Write BD	x		
HTTP Handle			x

Figura 16. Tabla que ilustra los contextos en los cuales se ejecuta cada operación.

En la fase de afinamiento, hubo un ciclo en el cual la aplicación era expuesta a pruebas de carga, y en base a eso los contextos eran afinados para optimizar el desempeño. Aquí se utilizó la herramienta de monitoreo *NewRelic*, con el fin de revisar el consumo de recursos y realizar perfilamiento de los hilos – lo cual ayudo a detectar cuellos de botella. En particular, la primera selección intuitiva de hilos se realizó en el *JDBCConnectionPool*, y por consiguiente en el *db-access-context*. Específicamente la base de datos tiene un límite de conexiones basado en el tamaño de la máquina. Nótese en la figura 14, que el número de hilos en los contextos involucrados con la conexión a base de datos, son iguales al límite de conexiones – ósea 24 en este caso. Esta decisión, se justifica en el hecho de que no tiene sentido tener grupos con más de 24 hilos. Por ejemplo, si se fija un número de 26 hilos, van a haber 2 hilos esperando siempre, ya que solo 24 pueden realizar consultas al tiempo. A diferencia del *db-access-context*, los otros dos no dependen del límite impuesto por la base de datos. En consecuencia la única manera de afinarlos es por medio de iteraciones.

6 Validación y pruebas

En la etapa de validación y pruebas, se realizaron los despliegues de cada arquitectura, para luego realizar pruebas de carga a cada una de las funcionalidades. Para lograr esto, se utilizó un cliente Apache JMeter que ejecutaba en una máquina de 16 GB de RAM y 8 cores. El objetivo de las pruebas era caracterizar el desempeño de cada arquitectura. Cabe recalcar, que las pruebas se realizaron en un requerimiento a la vez, es decir, cada funcionalidad tenía los recursos del servidor completamente disponibles para su ejecución.

Para caracterizar las arquitecturas, el objetivo era determinar el número de usuarios concurrentes que podía atender la aplicación, manteniendo tiempos de respuesta estables. Para esto se realizaron iteraciones en las cuales habían dos criterios para considerar falla. El primero determinaba que la prueba fallaba si el porcentaje de errores era mayor a 0%. Por otro lado, el segundo criterio indicaba que la prueba fallaba si el tiempo de respuesta evidenciaba un alza inesperada. Esta última, se basó en intuición a partir de la media y desviación del tiempo de respuesta.

6.1 Escenarios de prueba

	Number of DB records	Ramp up [segundos]	Loop count
Obtener Carritos	10000	60	1
Obtener Productos Más Vendidos	50000	60	1
Crear Producto	0	60	1
Crear Carrito	0	60	1

6.2 Resultados de desempeño

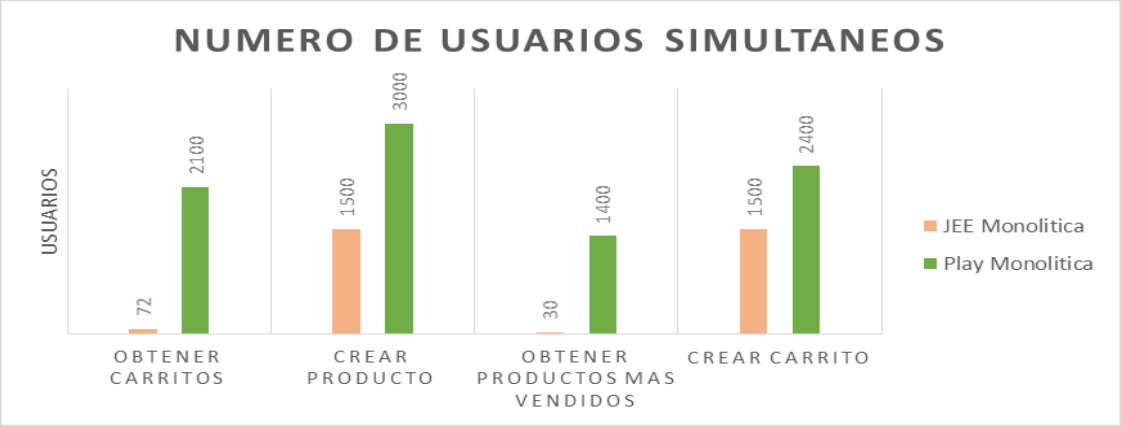

Figura 17. Número de usuarios concurrentes soportados por cada caso de uso.

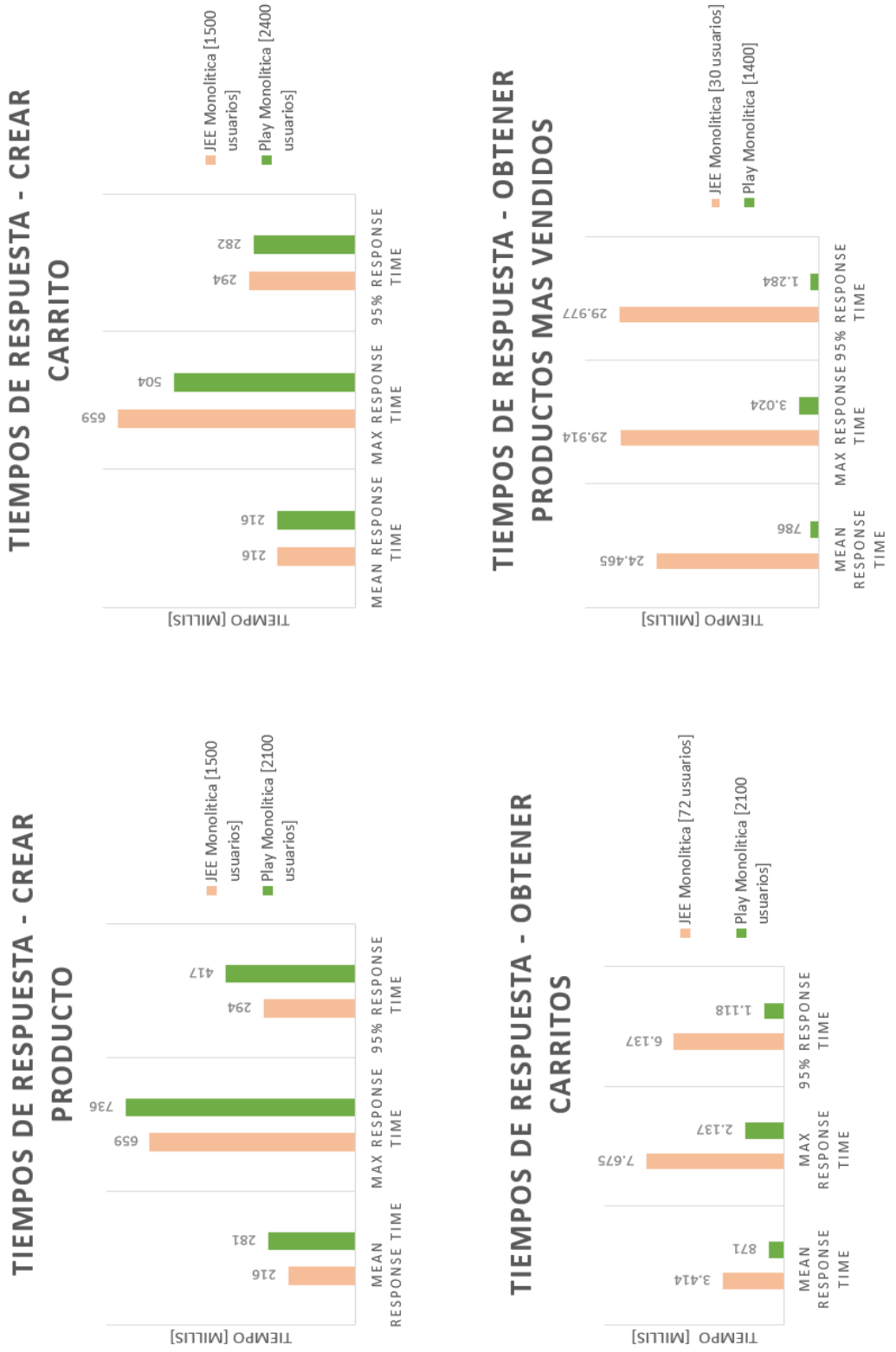

Figura 18. Tiempos de respuesta obtenidos por cada funcionalidad.

6.3 Análisis y observaciones

Los resultados de las pruebas permiten evidenciar el mejor desempeño de la arquitectura Play con respecto a la arquitectura JEE. Específicamente, se puede ver que en todas las pruebas el estilo de actores muestra un desempeño superior al estilo REST.

Con respecto a las pruebas que involucraban escritura en la base de datos, es posible ver que el desempeño entre las dos es comparable, porque las métricas del tiempo de respuesta son parecidas y las diferencias entre las peticiones que soportan no son alarmantes. Por ejemplo, el requerimiento de Crear Carito muestra que Play soporta 2400 peticiones, mientras que JEE soporta 1500. Además, en este caso los tiempos de respuesta son prácticamente iguales, ya que la métrica de 95% Response Time no difiere en más de 12 milisegundos. A pesar de esto, el estilo de actores logra prevalecer.

Por su parte los requerimientos de lectura, mostraron diferencias considerables entre las dos arquitecturas. En ambos se puede notar que Play soporta por lo menos 40 veces más carga que la aplicación JEE. Hay que destacar la diferencia abismal entre los tiempos de respuesta del requerimiento intensivo. Se puede ver que el estilo REST responde en alrededor de 30 segundos, mientras que la arquitectura basada en actores maneja tiempos menores a los 3 segundos. Esto último se puede explicar teniendo en cuenta que en Play fue posible aplicar un contexto de ejecución dedicado exclusivamente a procesar el algoritmo intensivo, y por esta razón se notan diferencias notables en el desempeño.

A partir de los análisis realizados, es evidente que los efectos del bloqueo son altamente nocivos para la arquitectura JEE monolítica. Esto se concluye partiendo de la gran diferencia en desempeños que tuvo en lectura y escritura. En este caso, los tiempos de bloqueo son mayores en las consultas de lectura, produciendo cuellos de botella nocivos para el desempeño. Caso contrario ocurre en los servicios de escritura, ya que las consultas son rápidas y por lo tanto los tiempos de bloqueo son menores.

En contraparte, la arquitectura de Play se vio altamente beneficiada por las tácticas de paralelismo, ya que permitieron mantener un sistema con más hilos para compensar las operaciones bloqueantes. Además de esto, el soporte en las funcionalidades asíncronas de Java 8, ayudaron a evitar la retención de recursos en el sistema.

7 Conclusiones y trabajo futuro

Este proyecto de grado, implementa una serie de tácticas aterrizadas a la plataforma Play Java, que permiten implementar código concurrente usando el modelo de actores. A partir de esta experiencia, se presentan una serie de prácticas que hacen posible obtener desempeños superiores en Play. También se logra predecir que la plataforma JEE depende de máquinas grandes para funcionar eficientemente, ya que el contenedor empresarial junto con el servidor, consumen una gran cantidad de recursos en tiempo de ejecución. En el caso de Play esto es completamente opuesto, debido a que la arquitectura está basada en el modelo de actores, que es bastante ligero en memoria.

Durante la investigación se descubrió que las aplicaciones Play incurren en un *trade-off* que involucra desempeño y mantenibilidad. Este último se da porque las mejoras en desempeño (i.e. contextos dedicados para cada operación) se mezclan en con la lógica de negocio, lo cual hace que el código se mas difícil de mantener. En esto último JEE es superior, ya que este se encarga implementar la concurrencia y permite parametrizarla en un archivo de configuración.

Como trabajo futuro, se propone la implementación de anotaciones en Play que permitan desacoplar las preocupaciones de desempeño, de tal manera que los desarrolladores solo se enfoquen en la lógica de negocio.

8 Bibliografía

Akka. (s.f.). *http://doc.akka.io/docs/akka-modules/1.3.1/modules/camel.html*. Obtenido de Camel - Akka Modules Documentation: http://doc.akka.io/docs/akka-modules/1.3.1/modules/camel.html

Aparicio, O. K. (2015). *Análisis de elasticidad y tolerancia a fallos.* Bogota: Universidade de Los Andes - Trabajo de grado: Magister en Ing Sistemas y Computacion.

Bass, L. (2013). *Software Arquitecture in Practice Third Edition.* Addison-Wesley.

Foy, J. R. (2014). *Play Framework Essentials.* PACKT Publishing.

Gupta, M. K. (2012). *Akka Essentials.* PACKT Publishing.

JUNEJA, V. (2015). Reactive Frameworks, Microservices, Docker and Other Necessities for Scalable Cloud Native Applications. *THE NEW STACK*.

Mario Villamizar, O. G. (2015). *Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud.* Bogota, Colombia: Systems and Computing Engineering Department.

Peschlow, P. (07 de December de 2012). *The H Developer*. Obtenido de The fork/join framework in Java 7: http://www.h-online.com/developer/features/The-fork-join-framework-in-Java-7-1762357.html

9 Anexos

9.1 Pasos para desplegar la aplicación Play en Heroku

1. En el archivo de configuración *conf/application.conf*, se debe cambiar el Secret que viene por defecto (i.e. changeme). En caso de que este no sea cambiado, la aplicación no se inicia en producción.

```
## Secret key
# http://www.playframework.com/documentation/latest/ApplicationSecret
# ~~~~~
# The secret key is used to sign Play's session cookie.
# This must be changed for production, but we don't recommend you change it in this file.
play.crypto.secret="caritopass"
#play.crypto.secret=${?APPLICATION_SECRET}
```

2. En el archivo de configuración se debe definir el data source a utilizar. Además, se debe configurar el manejador de persistencia Ebean, indicándole la carpeta que contiene los modelos. Para mayor reusabilidad se utiliza la variable de entorno JDBC_URL.

```
## JDBC Datasource
#
# Default database configuration using PostgreSQL database engine
#db.default.driver=org.postgresql.Driver
#db.default.url="jdbc:postgresql://127.0.0.1:5432/myPlayDb"
#db.default.username = "postgres"
#db.default.password = "caracoli"
#db.default.logSql=true

## Production heroku config db east
db.default.driver=org.postgresql.Driver
db.default.url=${?JDBC_URL}
db.default.username = "paotoya757"
db.default.password = ""
db.default.logSql=false


#
# Ebean configuration
#
ebean.default = ["models.*"]
```

3. En el archivo *conf/build.sbt*, se debe agregar la dependencia al driver de postrges.

```
libraryDependencies ++= Seq(
  javaJdbc,
  cache,
  javaWs,
  "org.postgresql" % "postgresql" % "9.4-1206-jdbc42"
)
```

4. En el root del proyecto se debe crear el archivo *Procfile*, pasandole el siguiente comando:

```
web: target/universal/stage/bin/reactive-mono-mktplace -Dhttp.port=$PORT
```

5. Por ultimo se debe desplegar la aplicación a heroku usando la siguiente secuencia de comandos desde el root del proyecto:

```
heroku create

git add .

git commit –m “Mensaje de commit”

git push heroku master
```

PROGRAMAS NECESARIOS:

- Git client para windows o osx
- Heroku toolbelt