**Arquitectura y diseño de software**

**Entrega experimento 1**

María Arévalo - 201415326

Sofia Hernandez - 20141410

Héctor Calderón - 201413725

**Pre-experimentación**

1. **Problemática:**

OilCol es una empresa colombiana de petróleos que cuenta con aproximadamente 1200 pozos de crudo. Actualmente cuenta con un monitoreo de estos de manera manual, por parte de los operarios. Debido a que este proceso es propenso a errores, la empresa desea automatizarlo mediante sensores y un sistema de procesamiento de la información recopilada.

1. **Objetivo del experimento:**

Para el desarrollar la aplicación Oilcol se escogió una arquitectura de tipo reactiva, usando el framework de desarrollo Play. Esta decisión se tomó debido a que consideramos que al tener este tipo de arquitectura, junto con el patrón de MVC usando dispatchers, es la manera más eficiente de solucionar los requerimientos funcionales y no funcionales del proyecto.

**3. Descripción del experimento:**

Para el experimento se va a someter la aplicación a las diferentes pruebas para verificar que cumple las peticiones básicas de CRUD, con el número máximo de usuarios que exigen los requerimientos. Se van a simular los datos que los sensores envían a la aplicación.

**4. Artefactos a construir:**

Para la solución se va a construir la aplicación Oilcol usando con arquitectura de tipo reactiva,asíncrona mediante el patrón MVC. Esto será desarrollado en Play Framework utilizando Intellij IDEA. Para simular la base de datos en esta primera entrega se utilizaron mocks.

**5. Recursos de la experimentación:**

La aplicación será corrida en una máquina virtual con 8gb RAM, con un sistema operativo de 64bits y Windows 7 Enterprise. Las pruebas serán ejecutadas con Apache JMeter 3.0. Para el despliegue de la aplicación se utilizará Intellij IDEA 2016.2 ultimate.

**6. Resultados esperados:**

Como se dijo anteriormente, se va a utilizar una arquitectura asíncrona, reactiva con el uso MVC y akka. Se escogió de esta manera debido a que podemos tener una aplicación que maneja las peticiones de manera concurrente y asíncrona, mediante el uso de FutureTask. Con esto, las tareas pueden ser pausadas y reanudadas, permitiendo que se puedan reusar los hilos de ejecución para otras tareas, una mejora considerable respecto a JAX-RS, ya que disminuye los bloqueos en la ejecución de la aplicación. Gracias a esta implementación, se verá mejorado el desempeño de la aplicación en cuanto a concurrencia, disminución de error, y tiempo de respuesta, apuntando directamente a los atributos de calidad de desempeño y escalabilidad.

**7. Duración y etapas:**

Para esta etapa se entrega la parte lógica de la aplicación implementada con mocks. Esto cumple los requerimientos base CRUD de cada entidad solicitada.

**Experimentación**

**Post-experimentación**

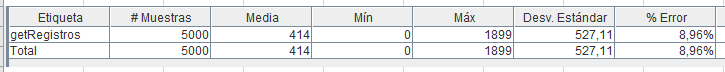
**Resultados obtenidos:**

Se adjunta un excel con los resultados obtenidos y gráficas correspondientes

**Artefactos construidos:** Todos los artefactos planeados fueron construidos. La aplicación Oilcol con mocks simulando la base de datos, corriendo en intellij y las pruebas en JMeter 3.0.

**Análisis:**

Recepción de información de 4800 sensores en ventana de 1 segundo:



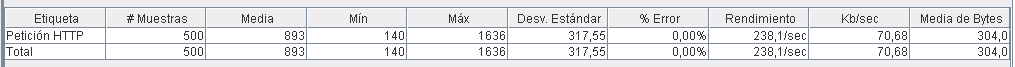
En este caso se simuló en JMeter el hecho de que llegaran 5000 registros de sensores en 1 segundo. Como se puede evidenciar, se obtuvo un error del 8,96% y una latencia media de 414. Esto nos lleva a pensar que el hecho de haber utilizado mocks y no una base de datos, es decir, la capa de persistencia, hace que haya un porcentaje de error mayor. Por esto se espera que al desarrollar dicha capa, el porcentaje de error se mantenga en 0.

**Conclusiones:**

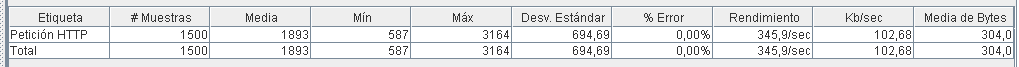
Debido a que en esta etapa solo se desarrolló la capa lógica, sin la persistencia, se obtuvieron porcentajes de error mayores al 0% y medias de tiempo que superan 1 segundo. Se espera que después de la implementación de la persistencia se vea mejorado el desempeño y la escalabilidad de la aplicación.

**Pruebas en JMeter con la capa de persistencia**

**Get Registros:** Número de muestras con 0% de error en un periodo de subida de 1 segundo.







**Get Registro:** Número de muestras con 0% de error en un periodo de subida de 1 segundo.







**Post Registros**



**Get Registros Fechas**



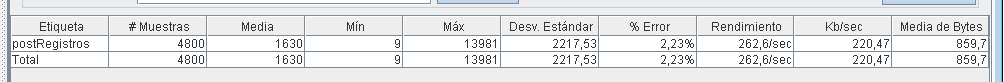
**Análisis**

El tiempo de respuesta aumentó con respecto al tiempo de respuesta con los mocks, esto debido a que ahora se está accediendo a una base de datos que no se encuentra en la misma máquina, por lo que el tiempo de latencia aumenta. Además, como los datos se están persistiendo, al momento de hacer get se están trayendo muchos más de los que se tenían con los mocks.

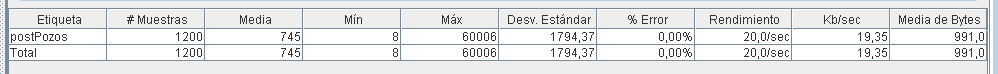
**Experimento 2:**

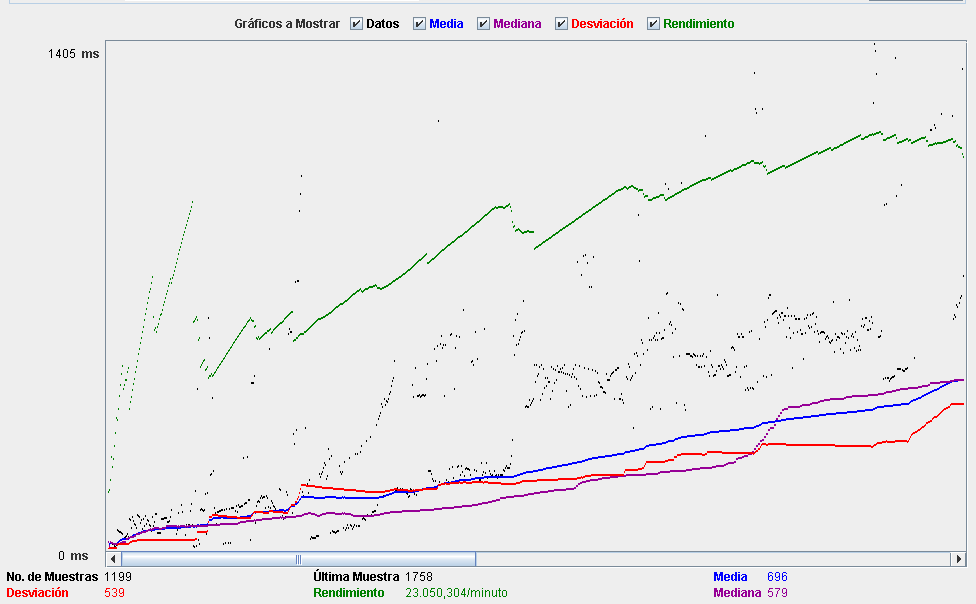
**PostRegistros:** Se registra la información proveniente de los sensores en una ventana de tiempo de 1 segundo

Con 4800 sensores



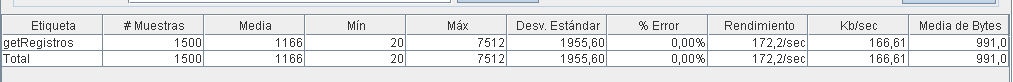
Con 1200 sensores

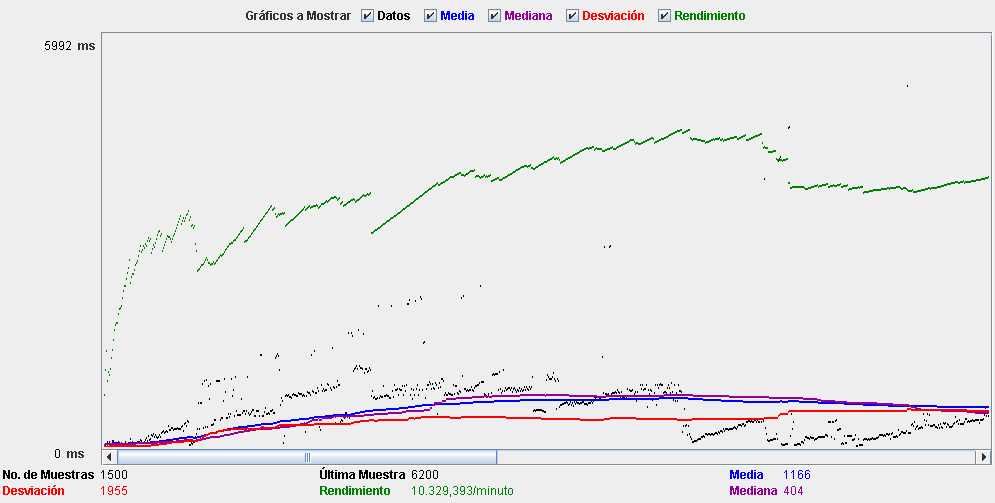




Para la entrega anterior el tiempo de respuesta para 1200 usuarios era menor que para esta, esto es por la latencia, como ahora la petición tiene que viajar más, esta aumenta, y aunque para este caso hayan 2 servidores, el tiempo de repuesta si se deteriora un poco, sin embargo, el número de peticiones que se pueden responder sin que haya error aumentó considerablemente, si bien antes podía con poco más de 1200 peticiones, ahora puede responder cerca de 4800 con 0% de error.

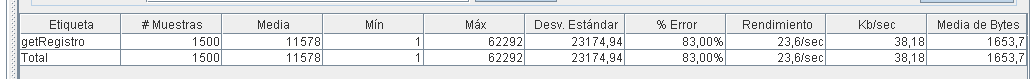
**GetRegistros:**

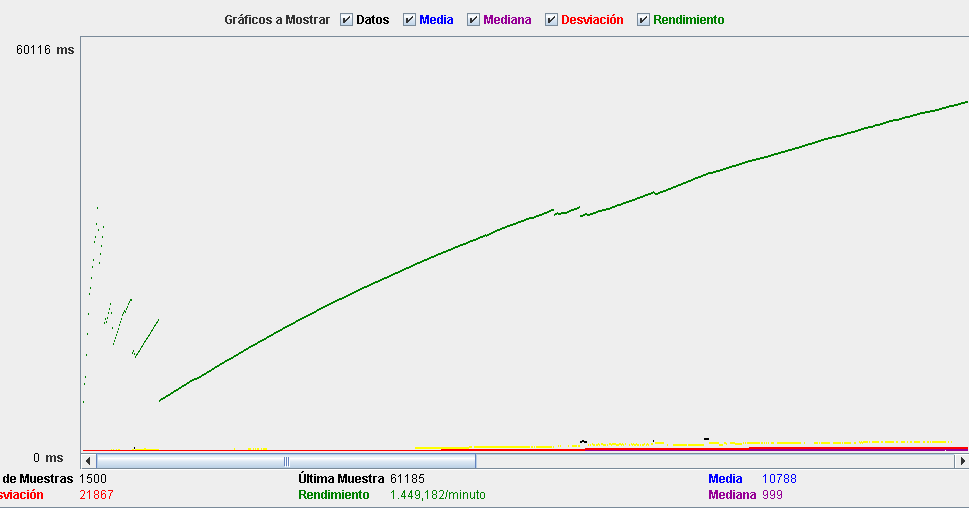




Como se observa, el tiempo de respuesta mejoró con respecto a la entrega anterior, sin balanceador de carga, en la que el número máximo aproximado de peticiones que se podían realizar con un periodo de subida de 1 segundo sin que hubieran errores eran 1200, ahora se observa que con 1200 se obtiene un mejor tiempo de respuesta, ya que hay un servidor adicional.

**GetRegistro:**



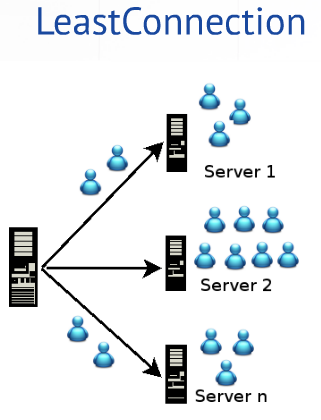


**Decisiones en la Arquitectura**

Para garantizar disponibilidad, se debe tener más de un servidor corriendo. Esto a razón de que un solo servidor puede ser muy sensible en el caso de tener muchas solicitudes concurrentes o bajo escenarios como un ataque DDoS, por ejemplo. Dejando esto en constancia, nace la pregunta de cómo direccionar las solicitudes a estos servidores. Por el momento y para el proyecto, se realizará por medio de un balanceador de carga básico, manejado por medio de la herramienta Nginx, donde no se están considerando ciertas variables que en un proyecto de mayor escala son de importancia (por ejemplo la geolocalización de los servidores y de las solicitudes).

La forma en que Nginx apoya la gestión de las solicitudes es en un servidor que no corre la aplicación, y es la IP a la cual se van a dirigir todas las solicitudes. Al recibir una de estas, Nginx intentará conectarse con los servidores que tienen la aplicación corriendo, si después de varios intentos no encuentra disponible al menos un servidor se dirige a sí mismo y se referencia como un servidor de emergencia, el cual, hasta este momento, genera un error de tipo 404 al cliente.

El balanceador de carga está configurado para que las conexiones se hagan de forma “Least Connection”. Esto significa que cada vez que llegue una nueva solicitud la va a redirigir al servidor disponible con menor carga. En la siguiente imagen se ejemplifica esto.



(Imagen “LeastConnection”, referenciado de https://es.wikipedia.org/wiki/Balanceador\_de\_carga)

Así, se utiliza un balanceador de carga para presentar un menor tiempo en la respuesta a solicitudes al no saturar un solo servidor y es necesario si deseamos contar con un indicador de mayor disponibilidad.

Adicionalmente, se desea terminar la implementación de una duplicación mantenida de la base de datos, en pro de mantener consistencia y disponibilidad en la información.