**Documento de experimentación**

**Pre-experimentación**

Problemática

La cantidad de pacientes, médicos y trabajadores del hospital no afecta el tiempo de respuesta frente a emergencias, toma de información cardiaca, sincronización con el dispositivo móvil, reconfiguración del marcapasos, envío de reportes al hospital y de consejos.

Objetivo

Probar que la cantidad de usuarios que reciben o envían información no afecta al tiempo de respuesta.

Descripción

Para llevar a cabo este experimento se diseñará el modelo conceptual del sistema, con base en la descripción del experimento, para luego construir la lógica de negocio, la cual contiene las funcionalidades que deben ser ofrecidas en el servidor del hospital. A continuación, se desarrollarán las pruebas de carga con las cuales se procede a verificar la escalabilidad y desempeño del sistema.

Artefactos a construir

Se construirá la lógica del sistema por medio de controladores y modelos haciendo uso de la arquitectura Play.

Recursos de la experimentación

Para llevar a cabo este experimento se usarán máquinas virtuales con sistema operativo de 64 bits. También se usará IntelliJ Idea Ultimate en su versión 2016.3.4

Para integrar el trabajo se usará un repositorio en Github y para probar los servicios se hará uso de Postman.

Además, se hará uso de la arquitectura Play, con el fin de manejar la concurrencia de eventos por medio de dispatchers.

Resultados esperados

Se espera que el sistema sea capaz de recibir información de frecuencia cardiaca, presión sanguínea y nivel de estrés, también debe poder recibir notificaciones de emergencias, consultar historial de un paciente, enviar consejos y reconfiguración de marcapasos, manejar la información de los pacientes junto con su historia clínica y debe soportar la recepción de información desde 3000 sensores en un rango de tiempo de 1 segundo.

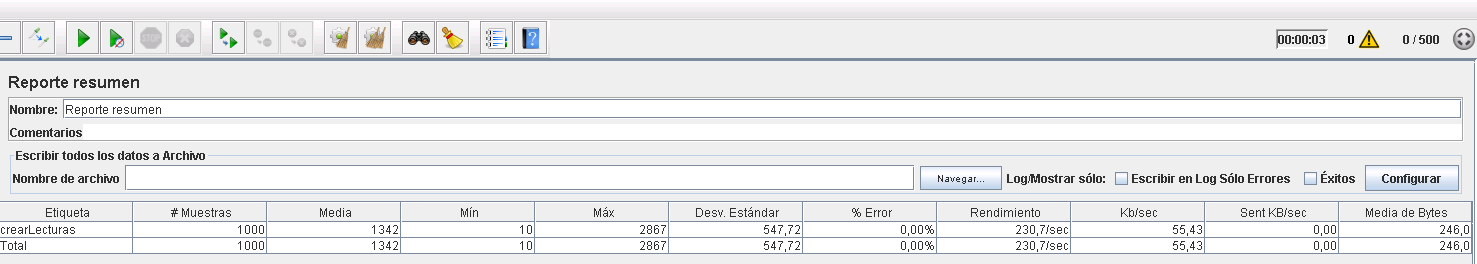
Duración y etapas

Se definen cuatro etapas. La primera es la etapa de análisis, en la cual se estudiará el enunciado del experimento y se definirán los requerimientos funcionales y no funcionales; esta etapa tomará una hora. La segunda etapa es la de diseño, en la cual se definirá el modelo conceptual y la arquitectura a utilizar; esta etapa durará tres horas. La tercera etapa es la de construcción de la lógica de negocio y tomará tres días. Por último, está la etapa de pruebas en la que se probará que el sistema soporte la recepción de información; esta etapa tomará tres horas.

**Experimentación**

**Post-experimentación**

Resultados obtenidos



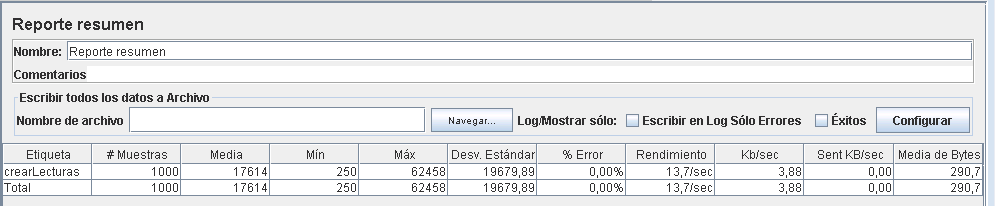
Los resultados obtenidos se pueden evidenciar a través de la tabla anterior, en la cual se puede observar que la media de respuesta es de 1342 milisegundos, con un rendimiento de 230,7 threads ejecutados por segundo con un 0% de error, lo que indica que todas las peticiones fueron atendidas exitosamente.



Adicionalmente, se puede ver mejor en la gráfica que el rendimiento fue mejorando cuando la cantidad de threads aumentaba hasta llegar a mil peticiones, además la desviación es baja lo que indica que la distancia entre los diferentes tiempos de respuesta no fue tan grande.

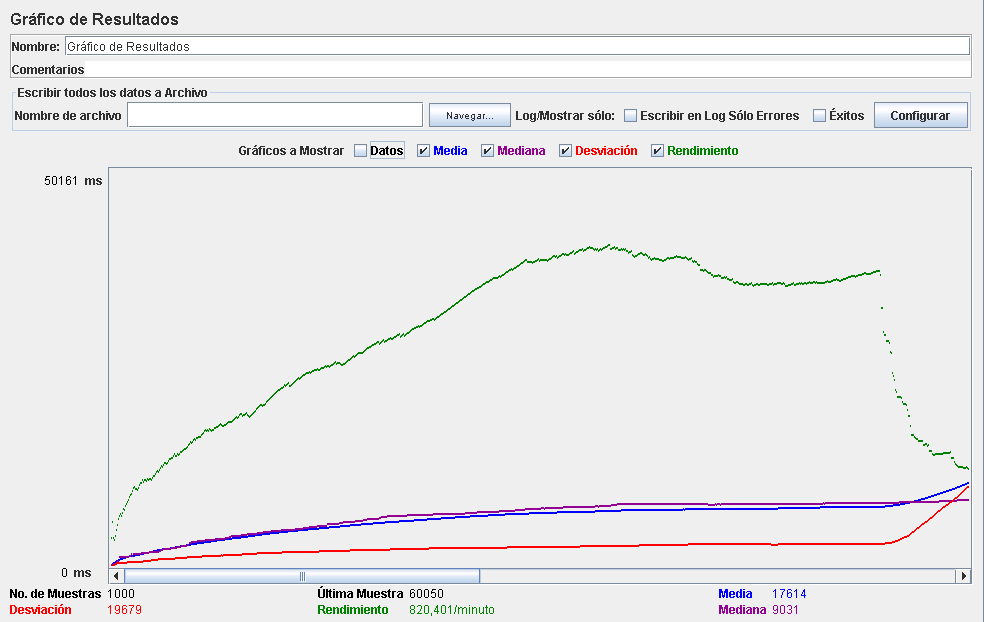
Resultados obtenidos con el balanceador de carga

Los resultados obtenidos al implementar el balanceador de carga, con el método Round Robin, se pueden observar a continuación:



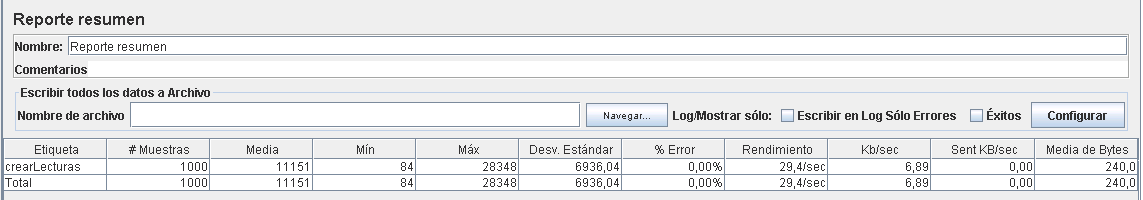
En esta se puede observar que la media fue de 17614 milisegundos, es decir, 17 segundos, con un rendimiento de 13,7 peticiones atendidas por segundo. Además, se observa que el porcentaje de error fue cero, por lo cual se sabe que todas las solicitudes enviadas fueron procesadas correctamente.

Adicionalmente, en la gráfica que se expone a continuación podemos ver la relación existente entre el tiempo y la cantidad de muestras. La línea verde muestra el rendimiento, se observa que a medida que aumenta la cantidad de muestras, en el tiempo más de ellas se van procesando. La línea azul nos indica la media, es decir la cantidad de tiempo que toma para responder con el número de muestras correspondientes, este tiempo va aumentando, pero no en gran medida, aumenta logarítmicamente. Es por esto que la línea roja, que expone la desviación estándar no aumente mucho, porque la diferencia de la media en cada punto no es tan grande.



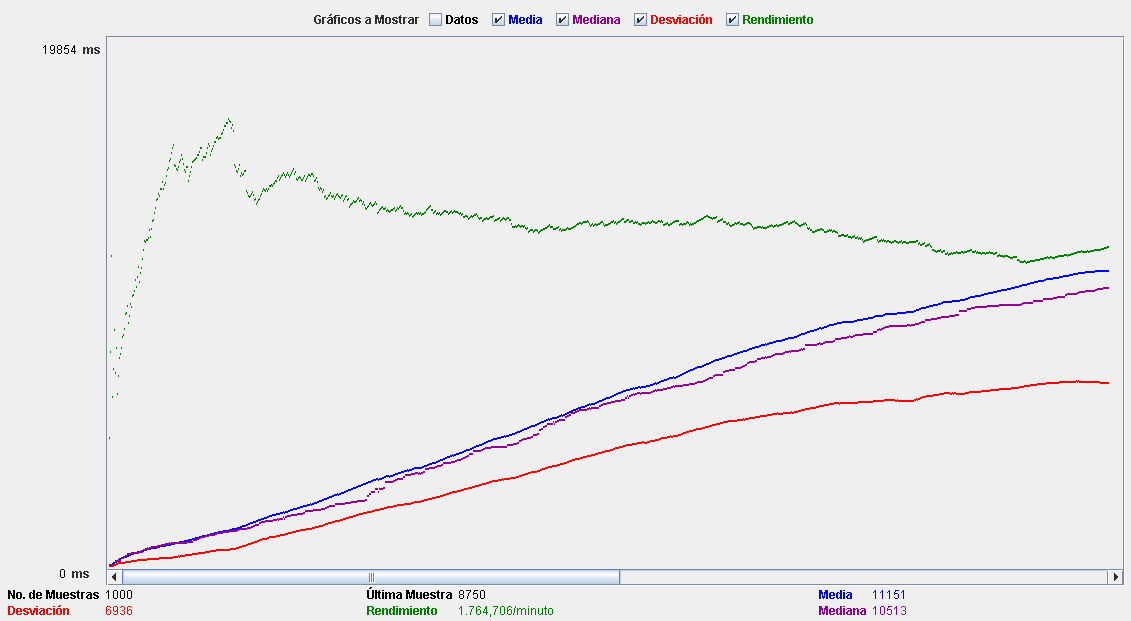
Resultados obtenidos con seguridad

Los resultados obtenidos el implementar algoritmos para asegurar confidencialidad e integridad se pueden observar a continuación:



En esta se puede observar que la media fue de 11151 milisegundos, es decir 11 segundos, con un rendimiento de 29,4 peticiones atendidas por segundo. Además, se observa que el porcentaje de error fue cero, por lo cual se sabe que todas las solicitudes enviadas fueron procesadas correctamente.

Adicionalmente, en la gráfica que se expone a continuación podemos ver la relación existente entre el tiempo y la cantidad de muestras. La línea verde muestra el rendimiento, se observa que a medida que aumenta la cantidad de muestras, se mantiene constante. La línea azul nos indica la media, es decir la cantidad de tiempo que toma para responder con el número de muestras correspondientes, este tiempo va aumentando.



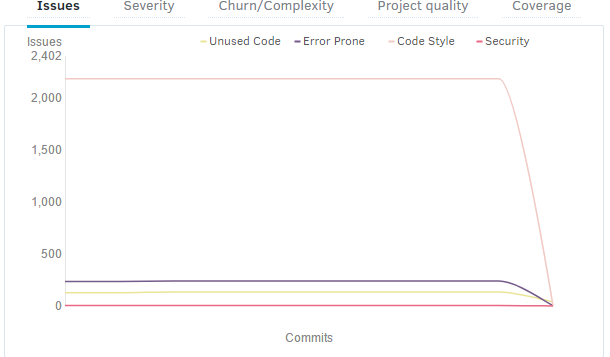
Duración real

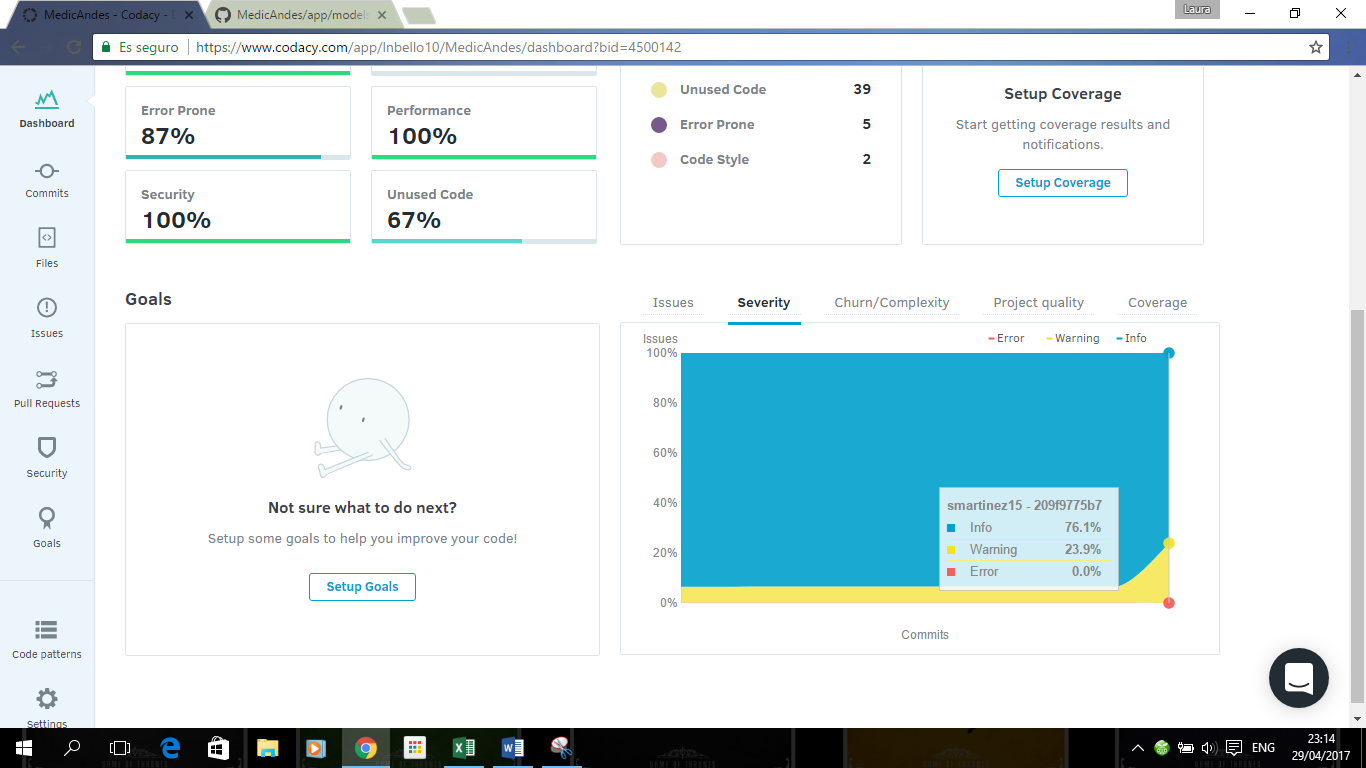
El tiempo teórico de análisis del enunciado del proyecto fue una hora y el tiempo experimental fue también de una hora. Por otro lado, en la segunda etapa el modelo conceptual y la arquitectura a utilizar se demoró 2:30 horas cuando el tiempo teórico fue de 3 horas. Además, la tercera etapa es la de construcción de la lógica de negocio que tomo 4 días cuando teóricamente se decidió que tomaría tres días. Finalmente, la cuarta etapa de pruebas nos llevó aproximadamente un día, cuando habíamos estipulado 3 horas.

Modificabilidad

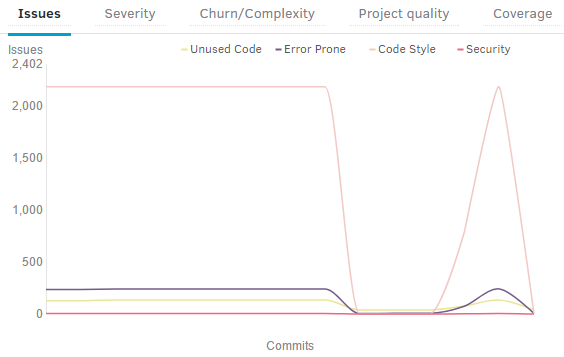
Según las métricas obtenidas en Codacy, antes de implementar los patrones de modificabilidad la clase que tenía mayor complejidad ciclomática, 25, era PacienteEntity. Dicha complejidad indica la complejidad lógica de un programa y ayuda a estimar el riesgo, costo y estabilidad. En este caso, la clase es compleja y es una clase de alto riesgo debido a que va a ser difícil probarla, entenderla y modificarla. Por otra parte, la clase que tiene más issues es UserController, con 8 issues, las cuales en su mayoría se deben a librerías importadas que no se usaron.

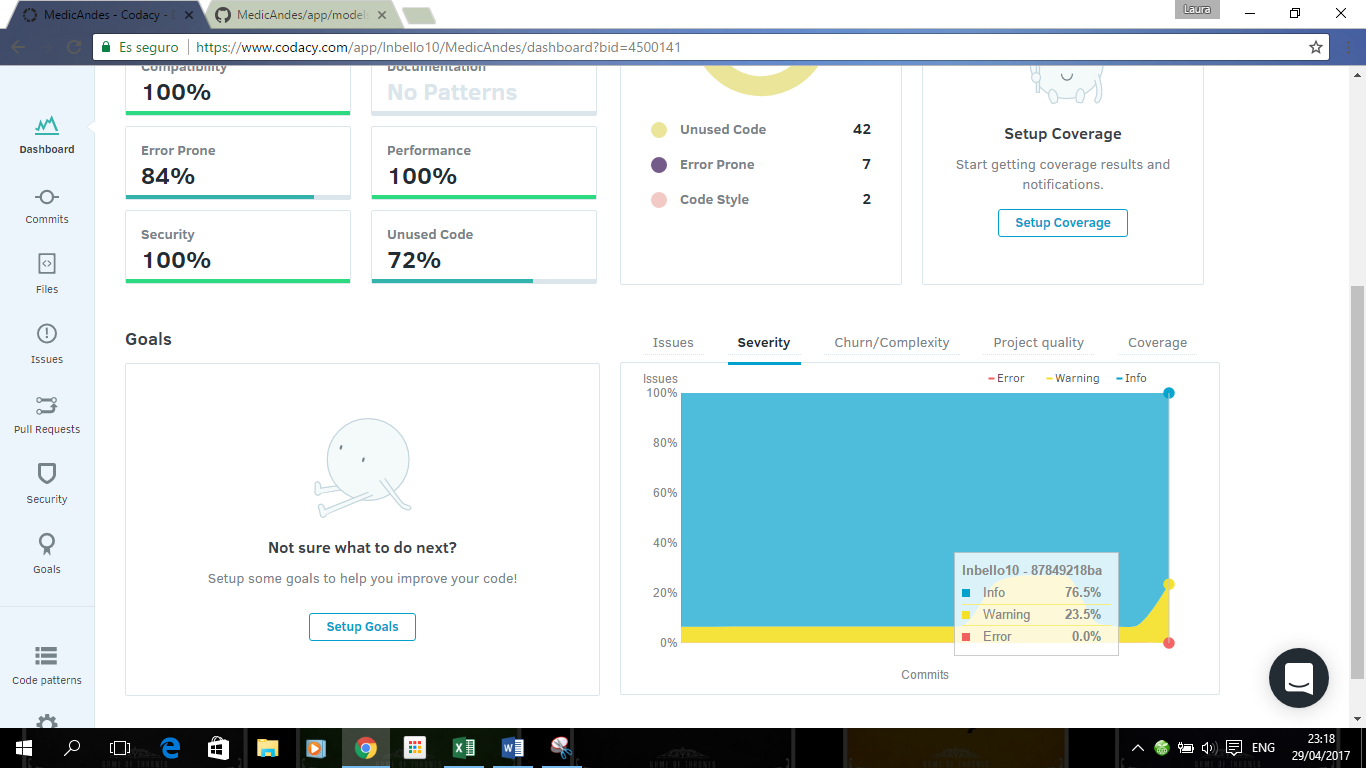
Antes de implementar modificabilidad se puede observar que la cantidad de issues disminuyó a medida que aumentaban los commits, esto dice que se fueron mejorando errores



Además, en la severidad se observa el % de errores, warnings(el % de warnings y errores indica que tan difícil será la mantenibilidad y si hay bugs escondidos) e info que se encontraron en el código:

Después de la modificabilidad



Los errores tuvieron picos, pero se logró arreglar.

Además, el % de warning disminuyó, lo cual evidencia que la modificabilidad ayudó.

En lo que respecta a otras métricas, se observa la comparación antes y después de implementar la modificabilidad:

|  |  |  |
| --- | --- | --- |
|  | Antes | Después |
| Code Style | 2 | 2 |
| Error prone | 5 | 7 |
| Unused code | 39 | 42 |
| Total issues | 46 | 51 |

La métrica code style hace referencia a las líneas de código que atentan contras las buenas prácticas, en este caso hubo dos clases a las cuáles no se les definió un paquete al que pertenecen.

Error prone dice que tan propenso es a errores, después de la modificabilidad aumentó en 2 líneas debido a que hay algunos métodos que pueden causar errores si no se saben usar, por ejemplo, usar un if…then…else para un método que retorna boolean.

Unused code da señales de cuantas líneas de código no fueron utilizadas en ningún momento, esto se debe en su mayoría a librerías que fueron importadas pero que no fueron usadas en ningún momento.

Se puede observar como el número de issues aumentó, no es un aumento significativo, pero da cuenta de que se pudo observar mejor cuales librerías eran realmente las necesarias.

Finalmente, el factor de acoplamiento es 1 antes ya que no había relaciones de herencia, pero luego de implementar la modificabilidad se añaden tres relaciones de herencia por lo que este factor disminuye. Las tres nuevas relaciones de herencia se deben al proxy del historial y a las dos nuevas clases que se crearon para los estados rojo y amarillo de la emergencia.

Patrones implementados

Proxy: Se implementó proxy debido a que el historial es un elemento que guarda muchas cosas y resulta costoso buscarlo en la base de datos cada vez que se quiere realizar una consulta, es por eso que se crea un proxy que guarda las últimas 15 lecturas en caso de que se quiera ver el historial de las lecturas. Adicionalmente, si alguien desea conocer el historial completo (citas, exámenes, lecturas, emergencias pasadas, alergias) este patrón ayuda a que solo se cree una vez el historial y se crea cuando es estrictamente necesario.

State: Este patrón dice que se cambia el comportamiento de un elemento cuando cambia su estado, es por esto que se decidió implementarlo para las lecturas. Una lectura recibe tres entradas, nivel de estrés, ritmo cardiaco y presión sanguínea, dependiendo de los valores que reciba la misma lectura va a cambiar su estado a verde, amarillo o rojo, es decir, hace una cosa diferente dependiendo de lo que lea de los sensores. Esto ayuda a que la lectura sepa cómo manejar la situación en tiempo de ejecución.

Observer: Cuando se implementa observer se asegura que un grupo de observadores están pendientes de lo que ocurre para comportarse de alguna forma en particular, es por esto que se implementó para la lectura. Al cambiar esta de estado, a amarillo, rojo o verde, se le notifica, dependiendo del caso, al médico y al sistema de emergencias. Ellos son los observadores de la lectura y los que van a tener algún tipo de comportamiento en especial, cuando se vuelve estado amarillo, el médico (que es un observador) es notificado y el comportamiento de él es enviar un consejo; cuando el estado es rojo se le notifica al sistema de emergencias y al médico para que vayan a recoger al paciente; finalmente cuando el estado es verde, no se realiza ninguna acción. Este patrón es diferente de state porque notifica a otras clases para que hagan algo, asigna responsabilidades, mientras que state cambia el comportamiento dentro de la misma clase.

Artefactos construidos

En nuestro proyecto implementamos el estilo de arquitectura por actores a través de Play, por esto manejamos dos paquetes principales, uno de controladores, y el otro de modelos (Entities). Estos paquetes se construyeron para manejar la lógica del negocio de manera que se acoplara bien con la tecnología de Play, además se hizo una conexión con la base de datos Postgres para manejar la persistencia y poder probar el funcionamiento del proyecto. No se implementó ninguna otra arquitectura ni artefacto, dado que solamente con estos paquetes se seguía el lineamiento de Play, manejo de concurrencia por medio de dispatchers.

Análisis

Uno de los motivos por los que el tiempo medio no fue mejor se puede deber a la conexión a la base de datos, dado a que toma mucho tiempo acceder a la memoria secundaria a recuperar los registros, y Play ya tiene la información en la RAM. Por otro lado, se usaron dos tácticas, una es la introducción a la concurrencia ya que se procesa en paralelo los eventos con threads, y el otro, mantener múltiples copias de computaciones, es decir, la introducción de dispatcher que controlan la división de la carga. De igual forma usamos FIFO en la táctica de Schedulling Policy para así, tratar las solicitudes como iguales, sin problema de que alguna tenga más prioridad que otra. Uno de los principales problemas es que en la prueba de carga se logra crear todos los threads en un segundo, pero toma más tiempo responder a las solicitudes ya que son solicitudes de POST, por lo tanto, la base de datos tiene que hacer revisión de restricciones, candados, entre otros.

Análisis comparativo

En los resultados obtenidos del experimento 1, se pudo observar que al ejecutar las pruebas desde una sola máquina la media era más baja que al ejecutarla cuando se implementó el balanceador de carga.

Con respecto a la diferencia de rendimiento, en el balanceador de carga se utilizó el método Round Robin, ya que es necesario repartir los POST de lectura entre todos los servidores de forma equitativa. En este método se le da a cada proceso un pequeño periodo de tiempo y luego se suspende para darle la oportunidad a otros procesos. Es por esto que se observa que el rendimiento es menor cuando se tiene el balanceador de carga. No se procesa totalmente una gran cantidad de solicitudes, sino que por segundo se van haciendo los procesos de a pocos. En el experimento 1 el rendimiento fue de 230,7/sec mientras que en el experimento 2 es de 13,7/sec.

Con respecto a la diferencia de medias, es probable que sea más alta luego de la implementación del balanceador debido a que algún servidor se pudo haber sobrecargado y como las solicitudes se envían de forma equitativa, el retraso de ese servidor afecta el tiempo total de respuesta a las solicitudes. Además, existe un tiempo adicional al momento de procesar las lecturas ya que estas se envían desde máquinas separadas, entonces eso toma un poco más de tiempo que cuando las lecturas se envían desde la misma máquina. La media en el experimento 1 fue de 1,3 segundos y en el experimento 2 de 17 segundos.

Por otra parte, los KB/sec es mayor cuando se ejecutaron las pruebas recibiendo solicitudes desde una sola máquina, eso quiere decir que un mayor número de información fue enviado. Mientras que con el balanceador este número es pequeño, se puede deber a que el tiempo destinado a los procesos es limitado, por lo tanto, la cantidad de datos que se envían también es limitado. En el experimento 1 fue de 55,43 y en el experimento 2 de 3,88.

Finalmente, en ambos experimentos el porcentaje de error fue cero, por lo que se puede concluir que en ambos el total de las solicitudes fueron atendidas, de forma diferente, pero todas se procesaron en su totalidad.

Análisis comparativo seguridad

Se cifró el envío de lecturas de manera que haya confidencialidad e integridad, al ir comparando con el hash obtenido localmente y el enviado en el mensaje.

Se ve una diferencia de medias, ya que seguridad es un trade off de desempeño. La media en el experimento 1 fue de 1,3 segundos y en el experimento 2 de 11 segundos. Lo cual es mejor que cuando se implementó el balanceador de carga. Se demora más cuando hay seguridad porque hay que descifrar la lectura que nos enviaron y eso toma un poco más de tiempo.

El rendimiento también se ve afecta, en este experimento fue de 29,4 y en el anterior fue de 230, esto se debe a que el descifrado retrasa un poco la creación de lecturas.

Se implementó autenticación y autorización, existen usuarios que tienen login, password y un permiso, el cual indica que tienen permitido hacer. Además, para poder acceder o modificar algo deben ingresar un token que se les asignará cuando hagan login, si este login no coincide con el usuario y la clave se lanzará un error, o si el token no es el mismo de la tabla, se lanza un error de autenticación.

Finalmente, en ambos experimentos el porcentaje de error fue cero, por lo que se puede concluir que en ambos el total de las solicitudes fueron atendidas, de forma diferente, pero todas se procesaron en su totalidad.

Conclusiones

En conclusión, la hipótesis planteada fue puesta a prueba y se obtuvieron resultados positivos, es decir, se comprobó que la cantidad de usuarios y solicitudes que recibía el sistema no tuvieron una alta influencia en la latencia (tiempo de respuesta entre el momento que se recibe la solicitud y en que se responde). Algunas recomendaciones son disminuir el acceso a la base de datos lo más que sea posible, también, es importante diseñar el modelo relacional muy bien desde el principio, para evitar que el nivel de acoplamiento sea alto.