

**Ingeniería en Sistemas**

**Computacionales**

**Raúl Tostado Blázquez I.D. 145901**

**Materia: Sistemas Distribuidos**

**Nombre del Trabajo: Práctica Websocket-Play-Actors**

**Profesora: Ofelia D. Cervantes Villagómez**

**San Andrés Cholula, Puebla a**

**06 de mayo de 2015**

Reporte práctica Websocket-Play-Actors

**Introducción a la práctica**

El objetivo de la actividad es desarrollar los métodos necesarios para implementar un chat anónimo utilizando el sistema de actores de Akka a través de Websockets con Play Framework.

El modelo de actores es un modelo matemático de computación concurrente que trata como "actores" a los principios de la computación concurrente: En respuesta a un mensaje que un actor recibe el actor puede tomar decisiones, como por ejemplo, crear actores, enviar mensajes y/o determinar cómo responder al siguiente mensaje. El modelo de actores adopta como filosofía que todo es un actor. Éste concepto es similar al de todo es un objeto utilizado en el paradigma orientado a objetos pero difiere en cuanto a que en un modelo orientado a objetos, el cómputo se ejecuta típicamente de manera secuencial, mientras que el modelo de actores es en esencia concurrente.

**Antes de comenzar:**

El primer paso fue dar una lectura detallada a la práctica para saber cuál es el contenido de la misma, posteriormente se descargaron como Zip los programas con los que se trabajó y otros complementos necesarios para trabajar.

**Introducción - Modelo de Actores**

En **el problema de los filósofos**, recordarás que escribir programas concurrentes no es una tarea sencilla. Tratar con **locks** y **threads** es una actividad en donde es altamente probable aparezcan errores, en muchos de los casos el código de aplicaciones concurrentes es difícil de leer, probar y mantener.

Es por esto que muchas personas prefieren evitar la concurrencia y en su lugar desarrollan algoritmos que se ejecutan en un **solo proceso secuencial**, confiando en servicios externos de terceros para administrar operaciones concurrentes asíncronas. La mayoría de las veces estas alternativas son viables, pero en algunos casos los sistemas requieren respuestas inmediatas y depender de terceros es una solución arriesgada.

Afortunadamente existe **Akka**. Akka es una librería integrada completamente con el lenguaje de programación **Scala**. Akka permite la construcción de aplicaciones concurrentes, distribuídas y tolerantes a fallas. La siguiente figura ilustra la comunicación entre actores, en donde podemos observar la comunicación entre actores mediante mensajes, además de la jerarquía de actores. La jerarquía de actores los hace tolerantes a fallas puesto que si un actor hijo llegara a fallar, este transmite la responsabilidad a su superior y así sucesivamente hasta llegar al root:

El modelo de actores en nuestra aplicación permite la concurrencia de mensajes, es decir, el servidor recibe mensajes de chat, asigna a cada usuario (**UserActor**) un id único secuencial, posteriormente concentra los mensajes en un único actor (**BoardActor**), el cual se encarga de enviar el mensaje a los demás actores y posteriormente los envía mediante WebSocket a los clientes. En el siguiente diagrama se ilustra el comportamiento de los **actores** al realizarse un envío de mensaje desde el cliente.

**Actividad 1 - Desarrollo Server-WebSocket**

El objetivo de la actividad es desarrollar los métodos necesarios para implementar un sistema de actores en el servidor. Deberás **analizar** y **complementar** el código de las clases que aquí se presentan dentro del proyecto incluído en este repositorio. A continuación vamos a desarrollar el código en el controlador que está definido en **controllers/Application.scala**.

Se genera un ID único para la sesión. La variable UID solicita el id de la sesión**request.session.get(UID)** en caso contrario (**getOrElse**) ejecuta la función anónima. En donde se genera un número consecutivo para la sesión mediante un contador.

Además, el método **(index)**, retorna una respuesta **HTTP (Ok)** en donde el método **OK** envía el valor**uid** a la vista en **Views.html.index** el valor de @uid.

**UserActor** define un actor, funciona como una puerta de enlace entre los actores conectados al Chat y al websocket. Este actor envía mensajes a **BoardActor.scala** y al **WebSocket**.

* Antes que nada **UserActor** se suscribe a **BoardActor**, así podemos informarle a **BoardActor**sobre éste nuevo actor y estará listo para recibir sus mensajes.

Como habrás analizado, los actores son entidades que se suscriben y reciben mensajes, cuando UserActor recibe un mensaje a través del WebSocket, crea un

Por último definimos el objeto **UserActor** el cual recibe un id y envía a **BoardActor** un JSON.

**BoardActor** es un actor que recibe mensajes desde cada UserActor y los envía a los demás. Después cada actor envía sus mensajes a través del WebSocket.

Primero debemos definir el comportamiento cuando el actor recibe un mensaje, cuando el mensaje es una subscripción BoardActor agrega al usuario en su lista (Set) y comienza a observar al actor. En caso de recibir un mensaje revisa su lista de usuarios y envía el mensaje a todos. Por último cuando un usuario termina su sesión el actor **BoardActor** lo retira de su lista.

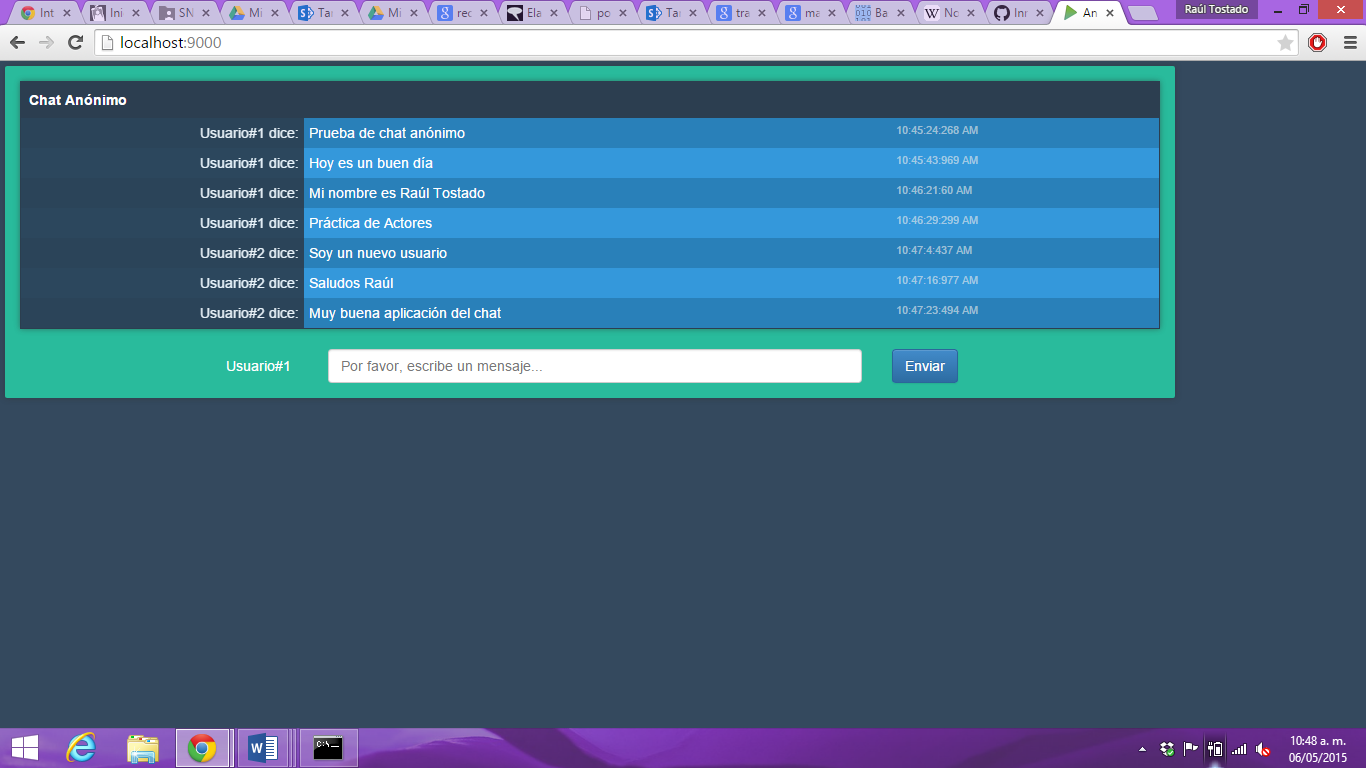
R= En esta primera parte de la práctica, se hizo un análisis detallado de cada una de las clases y se fue completando el código necesario para poder trabajar con el servidor

En resumen:

* Cuando se envía un mensaje se agrega el usuario al Set
* Cuanto se termina la sesión se remueve el actor del Set
* Cuando se suscribe un actor se agrega el usuario al Set

El **Chat Anónimo** ahora está completo, lo puedes usar desde diferentes navegadores para experimentar como **WebSocket** y **Akka** atienden las peticiones de todos los clientes. **Nota**, las sesiones de Play se mantienen en el navegador, por lo que para simular diferentes clientes deberás abrir otro navegador o ejecutar **Chrome** ó **Safari** en modo **incógnito**.

Por último agregaremos un timestamp, del momento en el que el servidor recibe el mensaje, para esto deberemos modificar nuestro **UserActor.scala** par que cuando se genere el objeto JSON agregue un campo con un timestamp.



R= Como se puede observar se tiene el chat funcionando, se hizo una prueba con dos usuarios en local host, uno en el navegador normal y el segundo desde el modo incógnito, también gracias a timestamp, fue posible registrar la hora en la que el usuario mandó el mensaje

**Actividad 2 - Limpieza**

La limpieza en este proyecto consiste en documentar el código en Scala, para esto deberás seguir las guías de ScalaDocs, puedes basarte en las descripciones presentadas en esta práctica.

* **Nombrar** adecuadamente las variables y métodos.
* **Agregar** un comentario antes de un método para indicar su funcionamiento.
* **Comentar** las líneas de código que podrían ser no muy claras.
* **Realizar** un identado adecuado del código.
* **Revisar** el código para detectar usos inadecuados de variables ó errores al escribir.

R= Esta parte se encuentra en la computadora en las clases utilizadas.

**Actividad 3 - Preguntas**

Responder ampliamente a las siguientes preguntas en el reporte, tomando en cuenta todas las actividades realizadas en este repositorio.

* **Explica** el modelo de actores

R= El modelo de actores es un modelo matemático de computación concurrente que trata como "actores" a los principios de la computación concurrente: En respuesta a un mensaje que un actor recibe el actor puede tomar decisiones. Los clientes son los actores que se van a conectar a un servidor para poder trabajar.

* ¿Un chat es una aplicación concurrente?

R= Si ya que cada vez que se conecta un nuevo actor o se manda un nuevo mensaje, es replicado a todos los usuarios

* Explica como harías un Chat sin el modelo de actores

R= Debería ser con Threads e hilos de ejecución, pero sería más complicado de implementar ya que hay que designar quien tiene los recursos para poder escribir un nuevo mensaje

* Explica como harías un Chat sin Websocket

R= Que cada determinado tiempo se actualice la página, la página va a guardar un registro de todos los mensajes nuevos, los cuales posteriormente serán visibles por todos los usuarios

* ¿Qué sucede cuando un actor recibe un mensaje?

R= Tiene la capacidad y obligación de replicarlo a los otros usuarios

* ¿Que representa **UserActor**?

R= Representa al usuario, el cual se conectará con el servidor

* ¿Por qué es necesario el uso de **WebSockets**?

R= Porque al ser una aplicación concurrente, es necesario que todos los usuarios vean las modificaciones en el momento que ocurren.

* ¿Cuál es el objetivo de **BoardActor**?

R= Tener un registro de todos los actores

* ¿Cuál es el significado de Future en el método **Application.ws**?

R= Nos dice si la comunicación con el servidor fue exitosa

* ¿En qué momento se ejecuta el método **Application.index**?
* Explica el uso de la variable **users** dentro de **BoardActor.scala**

**R=** Registra a los usuarios que están conectados con el servidor

**Conclusiones**

Gracias a la práctica, fuimos capaces de ver trabajar una aplicación concurrente, también pudo el salón de clases conectarse a un chat anónimo, demostrando que cada usuario manda nuevos mensajes.