EVOLUCIÓN Y ADAPTACIÓN DEL SOFTWARE

Reingeniería de Software – Agentes Móviles

David Villatobas Fernandez
GIS – Exp 141

1. Puesta en marcha

Se trata de un sistema que gestiona agentes móviles y sus comunicaciones.

Este sistema está compuesto de un servidor de dominio (“RaDomain”), y una o varias agencias (“RaAgency”).

El servidor de dominio se encarga de registrar las nuevas agencias y eliminar las que se cierren mediante el envío y recepción de mensajes. Almacena una lista de objetos RaAddress que básicamente son el host (nombre e IP) y el puerto por el que escucha cada agente.

La agencia se encarga de gestionar los agentes móviles, de modo que se puedan enviar y recibir agentes entre las mismas. Para ello, carga en memoria un agente, lo configura, y lo envía a otra agencia conocida, consultando previamente el servidor de dominio.

Al comenzar a tratar el código se ve claramente que están los ficheros desestructurados, por tanto hay que empaquetarlos en *packages* de manera correcta (raf.agentes, raf.config… etc). A continuación uno de los fallos más generalizados es el uso de *enum* como variable, ya que actualmente (al menos en java 8) pasa a ser una palabra reservada del lenguaje. Una vez limpio de errores, hay que establecer las rutas correctas para obtener las clases de los agentes y el fichero de configuración tanto de servidor de dominio como de las agencias.

Una vez configurado todo, al arrancar una agencia, se empezaban a mandar mensajes en bucle hasta terminar mostrando un error. Esto se debía a que la clase InetAddress ha cambiado y ahora el método *equals()* devuelve siempre *false* sea cual sea la entrada, por tanto, cuando recibe un mensaje y comprueba si ha llegado a su destino, siempre era falso, y reenviaba el mensaje una y otra vez. Se ha corregido usando simplemente el puerto ya que en la implementación que se desarrolle posteriormente no será necesaria la dirección IP.

En general se han solucionado algunos fallos sueltos de parseo de strings para que todo funcione correctamente.

2. Propuesta de cambio para el sistema

Inicialmente, mi propuesta fue una interfaz web para gestionar una agencia con Spring Boot. Una agencia sería un componente en Spring de modo que al arrancar el servidor mostrase en el navegador una lista de agentes disponibles para poder cargarlos, enviarlos y eliminarlos, es decir, ofrecer las mismas opciones que daba la interfaz java pero vía web.

Tras comprender la funcionalidad real del sistema y otros detalles al trabajar con Spring, la propuesta original ha sufrido unos cuantos cambios.

Por un lado se ha realizado una interfaz de usuario web para manejar una agencia. Por tanto podemos hablar de *frontend* mediante Angular 2 y TypeScript. Esta interfaz permite gestionar los agentes de una agencia ubicada en el servidor pudiendo cargar y enviar agentes y visualizando su respuesta mediante una consola.

Por otro lado se han adaptado tanto la agencia como el servidor de dominio para que funcionen como componentes de Spring Boot en el *backend*. Inicialmente pensé en generar plantillas html con Mustache, pero vi más rápido implementar un *frontend* sencillo con Angular que se interactuara con el servidor mediante un API Rest. Digamos que, al realizarse las implementaciones por separado, es mucho más sencillo desarrollar y testear cada componente.

3. Desarrollo del cambio para el sistema

1. Backend

En primer lugar he desarrollado el servidor API Rest. Este servidor está compuesto por dos componentes Spring *(@Component*), que se corresponden con la agencia y el servidor de dominio, y un controlador Rest *(@RestController*), que maneja las peticiones al API Rest.

Spring dispone de un sistema de inyección de dependencias interno. Es el propio framework el que crea cada uno de los módulos de la aplicación al iniciarla e inyecta las dependencias en los módulos que lo requieran. Al añadir la anotación *@Component* en una clase se considera un componente para el servidor. Para generar esa dependencia se declara en otro componente una variable de esa misma clase con la anotación *@Autowired.* De este modo, al arrancar el servidor, automáticamente instancia un objeto de esa clase y estará disponible para la clase en la que se declaró. En este caso concreto se declara en el componente *RaController* de la siguiente manera:

```
@RestController
public class RaController {

    @Autowired
    RaDomainComponent domain;

    @Autowired
    AgencyComponent agencia;
```

Al realizar esta configuración cada uno de los componentes *domain* y *agencia* estarán disponibles para cualquier petición del servidor y serán únicos. Para el caso del servidor de dominio tiene mucho sentido ya que se trata de que ese componente se pueda consultar en cualquier momento por cualquier agencia. Sin embargo una agencia compartida para varios clientes web no es que sea realmente útil.

Por ello, es necesario añadir la siguiente anotación al componente de la agencia:

```
@Component
@SessionScope
public class AgencyComponent
```

Al añadir la anotación *@SessionScope* se instancia un componente para cada sesión de usuario. Esto quiere decir que cada vez que se haga una petición web que se considere una nueva sesión (no basta con actualizar el navegador) se genera una nueva agencia en el servidor.

Por todo esto, la configuración de cada componente agencia se limita a autogenerar un puerto diferente para cada nuevo componente, dando igual la dirección IP ya que se encuentran todas las agencias en el mismo *host.*

Inicialmente pensé usar el *System.out* de Java para redirigir la salida de la consola nativa a una variable y poder enviarla mediante el API Rest al cliente. Pero tratándose de un recurso del sistema al que acceden concurrentemente todos los componentes que se ejecuten, resultaba muy tedioso separar la información que le corresponda a cada uno. Y mucho más si varias agencias actúan al mismo tiempo. Por ello he sustituido en el componente de la agencia el método *System.out.println()* por el método *agencyPrint().* De este modo la variable *log* va almacenando la actividad de su propia agencia, creando cierta independencia con el resto.

El componente *RaController* se encarga de gestionar las peticiones rest mediante métodos para recoger listado de agentes disponibles, agentes cargados y agencias arrancadas, y métodos para cargar, eliminar y enviar un agente.

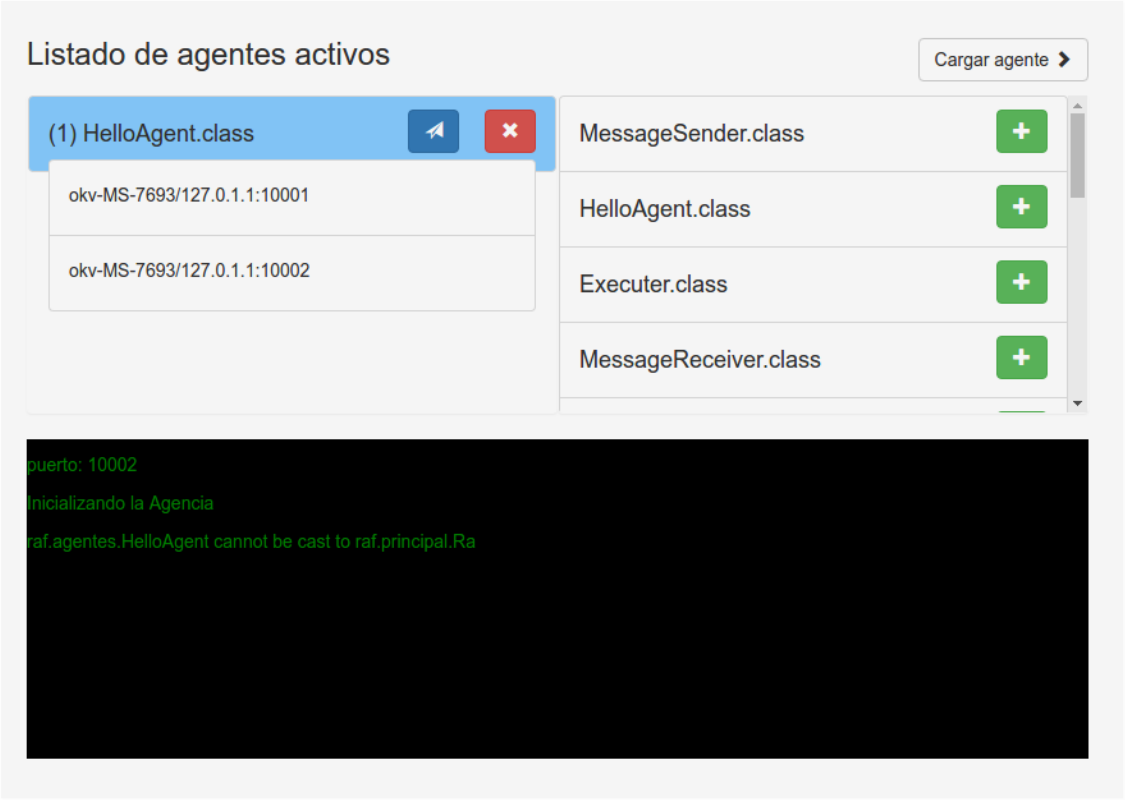
Cabe destacar que a la hora de cargar un agente da el siguiente error:

```
SendMessageThread: socket creado a: okv-MS-7693/127.0.1.1 10001
/127.0.0.1
SendMessageThread: Escrito mensaje en el socket.
java.lang.ClassCastException: raf.agentes.HelloAgent cannot be cast to raf.principal.Ra
ReceiveMessageThread: ha llegado un mensaje AGENCYS.
SendMessageThread: All closed
```

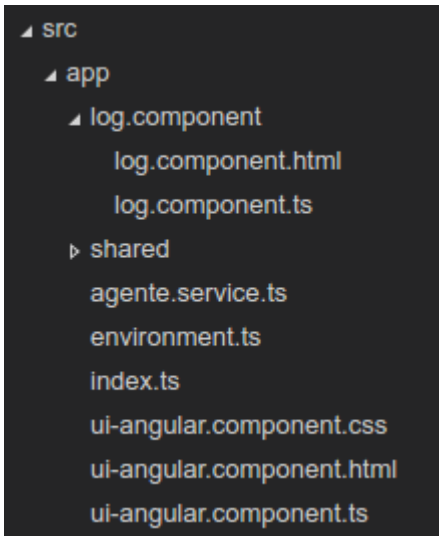
Tras mucho indagar en el tema no he encontrado explicación para que una clase que hereda de otra no pueda hacerse un *cast,* y más cuando fuera del proyecto de spring no da ningún problema. Cabe la posibilidad de que spring no sea compatible con alguna de las dependencias de alguna clase concreta del sistema, como por ejemplo con clases serializables. Es por ello por lo que no es posible realizar la gestión deseada con los agentes desde el entorno web hasta solucionar ese problema.

2. Frontend

Se trata de una implementación de lo más sencilla donde se busca la funcionalidad más que otra cosa:



La estructura seguida en Angular es muy sencilla:



Se divide en dos componentes, la aplicación principal y el log. Como la aplicación es tan sencilla opté por montar la plantilla y la lógica en el mismo componente por comodidad.

El log está en un componente separado porque cada pocos segundos lanza comprobaciones al servidor en busca de cambios en el mismo.

3. Conclusiones

El sistema resultante permite desplegar tantos agentes como clientes web queramos con las limitaciones lógicas de un servidor web. Dichas limitaciones serían extremadamente sencillas de resolver ya que con que exista un único servidor de dominio se pueden arrancar tantos servidores spring en diferentes servidores físicos como queramos.

Por otro lado, por optimizar el sistema, las comunicaciones dentro del mismo servidor spring se podrían hacer de un modo directo, obviando los sockets, de este modo se limitaría mucho el uso de recursos de red. Aunque esta optimización sería contraproducente en caso de querer escalar el sistema.

Finalmente habría que adaptar los agentes al entorno web, ya que un agente no sería capaz de lanzar una ventana Java.