E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

Rastreador de términos sobre redes sociales, incluido en el proyecto DNow

Grado en Ingeniería Informática

Trabajo Fin de Grado

Gontzal Aparicio Sampedro

Raúl Orduna Urrutia, Carlos López Molina

Pamplona, 27 de Junio de 2014

upna
Universidad
Pública de Navarra
Nafarroako
Unibertsitate Publikoa

Resumen

Este proyecto trata de la construcción de un modulo de recogida de información desde una serie de redes sociales a partir de palabras clave. El módulo forma parte de una aplicación desarrollada por el Diario de Navarra, dentro del proyecto DNow, para la generación de hilos de noticias y su posterior consumo en dispositivos móviles. El módulo debe encargarse de buscar información en una serie de redes sociales, así como de modelarla de manera homogénea para su puesta a disposición de un periodista, que seleccionará cuales de ellas son relevantes para los clientes suscriptores. El alumno ha sido encargado del proceso completo de análisis de requisitos, diseño del módulo y posterior implementación y prueba del mismo.

Keywords

Redes sociales, Periodismo, Recuperación de información, Búsqueda sintáctica, Conocimiento social, APIs de búsqueda, Programación sobre Cyclone, Gestión de Websockets.

Índice general

1. Introducción **7**
1.1. Contexto social en el que se desarrolla el trabajo . . . 7
1.2. Objetivos . . . 8
1.3. Definición de términos . . . 9

2. Requisitos y arquitectura **13**
2.1. Requisitos de desarrollo . . . 13
2.2. Requisitos de arquitectura lógica . . . 14
2.3. Requisitos de interfaz gráfica . . . 15
2.4. Requisitos de arquitectura física . . . 15

3. Análisis funcional **17**
3.1. Análisis de casos de uso . . . 17
3.2. Visión dinámica del sistema . . . 27
3.3. Diagrama de secuencia del cliente . . . 27

4. Diseño técnico **33**
4.1. Componentes del sistema . . . 33
4.2. Interfaces del sistema . . . 35
4.3. Diagrama de clases . . . 37
4.4. Protocolos de comunicación . . . 39
4.5. Formatos de intercambio . . . 40
4.6. Diagrama de despliegue . . . 42

5. Uso del sistema **45**
5.1. Guía de instalación . . . 45
5.2. Guía de uso básico del servidor . . . 45
5.3. Guía de usuario del cliente/periodista . . . 47

6. Problemas encontrados **51**
6.1. Tecnología de parte del servidor y tiempo-real en el cliente . . . 51
6.2. Problemas en los escuchadores . . . 52
6.2.1. Escuchador de Twitter . . . 52
6.2.2. Escuchador de Flickr . . . 53

6.2.3. Escuchador de Youtube 54
6.2.4. Escuchador de Facebook 55
6.3. Problemas en el uso de Embedly 56
6.4. Rastrear términos que incluyen ñ o acentos 57
6.5. Archivo de configuración 58

7. Pruebas del sistema **59**
7.1. Pruebas de integración 59
7.2. Pruebas de carga con ApacheBench 60

8. Resumen y Conclusiones **65**
8.1. Resumen 65
8.2. Conclusiones 66

9. Bibliografía **67**

Índice de figuras **70**

Capítulo 1

Introducción

En este documento se detalla el análisis, diseño y desarrollo del módulo para rastrear redes sociales perteneciente al proyecto DNow de Diario de Navarra.

1.1. Contexto social en el que se desarrolla el trabajo

En los últimos años hemos presenciado el despegue de ciertos medios de comunicación que se han dado a conocer como medios sociales. Estos medios han sido objeto de estudio desde diversas perspectivas, tanto sociales, como económicas y tecnológicas. De cualquier manera, se ha llegado a un entorno en el que casi cualquier ciudadano con acceso a internet es capaz de producir y consumir en tiempo real gran cantidad de información. Estamos, en definitiva, ante la consagración del paradigma de los productores-consumidores de información, que en los inicios de esta era tecnológica fueron bautizados como *prosumers*. Todo este conjunto de tecnologías y plataformas se han denominado *medios sociales* o *medios sociales colaborativos*.

Estos cambios tecnológicos han producido un impacto relevante en diversos ámbitos de la sociedad. En este trabajo nos centramos en su impacto en los medios de comunicación, pero es de justicia mencionar las grandes modificaciones que han sufrido tareas cotidianas en el ámbito social o cultural gracias a la aparición de medios sociales. El auge de los *medios sociales colaborativos* abre nuevas oportunidades para mejorar la actividad de las empresas que se dedican a la información. Dentro de este elenco de medios podríamos citar a los blogs, quizá la forma pionera de comunicación en esta tendencia, pero también a los espacios web de opinión, expresión y debate. Estos medios ofrecen plataformas baratas y sencillas que brindan a muchos individuos la posibilidad de acceder y comunicarse de forma masiva con otros. Provistos de estos medios, los ciudadanos han dejado de ser meros receptores de la información o los servicios, para convertirse en agentes activos de la vida pública y social.

Las empresas que se dedican a la información necesitan habilitar cauces para aprovechar todos los medios sociales accesibles en la red. Su reto no es solamente conseguir que estos cauces sean transparentes, accesibles, claros y sencillos, si no que deben encargarse de fusionarlos adecuadamente con sus fuentes/canales clásicas/os de información.

Es decir, deben sumar a la información accesible en los medios sociales aquella que es céntrica en su modelo de negocio (por ejemplo, el periódico en papel en los grupos editoriales), para estar allí donde se produce la conversación, allí donde sus clientes se expresan y debaten. Los medios sociales proporcionan información en tiempo real gracias al enorme flujo de mensajes, *tweets* o *posts* que se generan por minuto, siendo por ello altamente interactivos y dinámicos, generando conversación continua. Idealmente, los medios de comunicación serán capaces de unificar las noticias disponibles en las redes de comunicación tradicionales con aquellas sociales, para presentar a sus clientes una información lo más completa posible.

1.2. Objetivos

Diario de Navarra quiere adaptarse a esta nueva realidad y propone el uso de una aplicación móvil a través de la cual sus usuarios podrán acceder a información relacionada con temas de interés. En concreto, esta aplicación móvil debe proveer a los usuarios registrados de noticias acerca de ciertos temas seleccionados, obteniéndose estas a partir de diversas fuentes y medios. En este contexto es un periodista el que, trabajando desde el periódico, se encarga de buscar, filtrar y seleccionar noticias aparecidas en la red, y complementarlas con aquellas disponibles en el histórico del periódico. De esta manera, se intenta que sea el periodista y no el propio usuario el encargado de buscar, filtrar y combinar la información de las diferentes fuentes accesibles para cada temática.

El sistema a desarrollar está formado por varios componentes. Por un lado, tenemos la aplicación móvil a la cual se han de suministrar noticias específicas. Para cada noticia se crea un *hilo*, que es actualizado en directo por el periodista o por un equipo de periodistas en el Diario de Navarra. Dichos periodistas manejaran a su vez una aplicación web en la cual pueden tanto visualizar las noticias e informaciones accesibles (en el periódico y en diversos medios sociales), como añadirlas al hilo de la noticia. Consideraremos la aplicación móvil como el *frontend* del sistema, mientras que la aplicación en la que trabaja el periodista es el *frontal del periodista*. Cuando un periodista añada una noticia o información al hilo, será visible como una *notificación* dentro de la aplicación móvil del usuario. Estas notificaciones podrán ser de dos tipos: Adquiridas semánticamente o adquiridas a tiempo real de diversas redes sociales. Las semánticas serán extraidas de bases de datos ya formadas por el Diario de Navarra utilizando técnicas de búsqueda semántica. Aquellas obtenidas de redes sociales serán tomadas en tiempo real de diversas redes sociales de interés (Twitter, Facebook, Flickr, Google+,...).

El trabajo desarrollado dentro de este trabajo de fin de grado se centra en la adquisición de noticias dentro de las redes sociales. Mediante un servidor asíncrono, el periodista buscará en tiempo real en las principales redes sociales. Esto le proporcionará una serie de *tweets*, publicaciones, videos o fotos que estén relacionados/as con una serie de *tags* o temas que haya introducido para definir el hilo (o noticia). Dentro de esta colección de información, el periodista se encargará de filtrar la más relevante para incorporarla al hilo temático.

La implementación del sistema de información del periodista está desarrollado sobre

el *framework* web para Python *Cyclone*. Cuando un periodista realiza una petición al servidor, se establece una conexión de WebSocket entre el mismo y el cliente para proporcionar un flujo continuo de información de los medios sociales en tiempo real. Manteniendo las conexiones abiertas conseguiremos mantener actualizado al periodista desde los medios sociales sin necesidad de recargar la página. Debido a que se prevee un máximo de 20 periodistas al mismo tiempo y que el tráfico que genera cada uno no es excesivo, no existe problema en mantener las conexiones abiertas.

El objetivo prioritario de este trabajo de fin de grado es construir un motor de búsqueda en redes sociales que tenga la mayor precisión posible. Idealmente, el motor de búsqueda sería capaz de seleccionar, dentro de los medios sociales, todas (y sólo aquellas) noticias relacionadas con una temática, para aliviar el trabajo del periodista. Ademas, se han de facilitar las noticias obtenidas en un formato estandarizado para que el periodista se centre en el contenido y no en el diseño.

Teniendo esto en cuenta, los objetivos principales del proyecto son los que se describen a continuación:

- Construir un servidor asíncrono que responda a las peticiones de los usuarios, permitiendo elegir al usuario que escuchadores quiere activar cuando crea un hilo, que opciones han de tenerse en cuenta así como, obviamente, que palabra o palabras han de rastrear en sus respectivos medios sociales.
- Construir diversos escuchadores para las distintas redes sociales que se contemplen. Decidí priorizar los escuchadores a implementar de la siguiente manera: Twitter, Flickr, Youtube, Facebook, Google+ y Pinterest. Al final, se han implementado los cuatro primeros. Estos escuchadores deben buscar una serie de palabras clave y extraer la información de los datos devueltos por las redes sociales.
- Construir o encontrar un sistema que permita estandarizar los datos devueltos por la redes sociales para poder devolver/almacenar los datos obtenidos de una manera normalizada.
- Garantizar que el servidor, después de haber inicializado los escuchadores y obtener resultados, envía a los clientes activos en ese momento los documentos que haya conseguido de las redes sociales en un formato correcto para la vista y estandarizado.
- Establecer un sistema de conexión en tiempo real entre el cliente y el servidor para permitir la transmisión de datos o mensajes en ambos sentidos. Este sistema debe permitir que el cliente envíe mensajes al servidor sin necesidad de formularios y que pueda recibir datos del servidor sin necesidad de recargar la página.

1.3. Definición de términos

Hay una serie de conceptos y/o términos que se utilizarán frecuentemente a lo largo de este informe. Con el objetivo de facilitar su comprensión, se describen a continuación los más destacados:

- **Proyecto DNow:** Proyecto original de Diario de Navarra que consta de tres módulos generales. Un buscador en redes sociales, un buscador semántico y una aplicación móvil a la que se sirven los resultados.
- **Buscador en redes sociales:** Titulación formal de este proyecto, es decir, del módulo general que devuelve noticias en tiempo real, obtenidas de diversas redes sociales.
- **Buscador semántico:** Módulo que realiza una serie de búsquedas semánticas sobre la base de datos de Diario de Navarra. Se encuentra fuera del alcance de este proyecto.
- **Aplicación móvil:** Aplicación desarrollada para presentar a usuarios finales los resultados de ambos buscadores.
- **Palabras clave/Keywords:** Referencia a aquellas palabras que el periodista introduce en el sistema para que sean rastreadas en las redes sociales.
- **Escuchadores/Rastreadores:** Referencia a los módulos del sistema que se encargan de buscar en las redes sociales los *tweets*, *posts*, videos o fotos que tengan relación con las palabras clave introducidas por el usuario.
- **Operario:** Empleado de Diario de Navarra, preferentemente con conocimientos de informática, que se encarga de arrancar y apagar el sistema.
- **Periodista/Cliente:** Empleado de Diario de Navarra que se encarga de filtrar las noticias que le lleguen a la pantalla y enviar las que sean las más adecuadas a la aplicación móvil.
- **Python:** Lenguaje de programación en el que se ha codificado la parte del servidor.
- **Websocket:** Protocolo de comunicación que habilita un canal bidireccional de comunicación entre el servidor y el cliente, permitiendo el flujo de mensajes en tiempo real.
- **Tornado:** Framework web para Python que proporciona una serie de librerías asíncronas ideadas para colaborar con Websocket.
- **Twisted:** Framework de red para Python diseñado para facilitar la programación dirigida por eventos.
- **Cyclone:** Framework web para Python que combina los mejores aspectos de Tornado con los de Twisted.
- **HTML:** Lenguaje de etiquetado que es utilizado para realizar páginas web.
- **API:** Conjunto de funciones y procedimientos que ofrece una biblioteca externa para ser utilizadas por otro *software*.

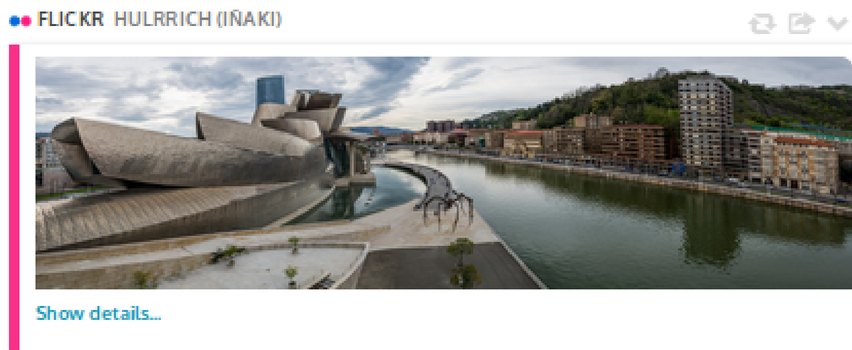

Figura 1.1: Ejemplo de tarjeta HTML.

- **Tarjetas HTML:** Concepto que referencia a las noticias que devuelven las APIs de diversas redes sociales tras haber sido formateadas a un aspecto estandarizado. Un ejemplo de tarjeta HTML puede apreciarse en la Fig. 1.1.
- **Embedly:** Conjunto de herramientas en linea que permiten construir tarjetas HTML con llamadas a su API.

Capítulo 2

Requisitos y arquitectura

A continuación se detallan la captación de los requisitos de desarrollo, de arquitectura lógica y física y de interfaz gráfica que forman parte de este proyecto.

2.1. Requisitos de desarrollo

Los requisitos de desarrollo describen que funcionalidades tiene que proporcionar el sistema para considerarse correcto.

- Los periodistas pueden iniciar una serie de escuchadores de redes sociales:
  - Mediante un formulario, el sistema debe proporcionar al periodista que escuchadores puede activar, las opciones que se puedan incorporar a dichos escuchadores y un campo para introducir las palabras clave que desee buscar.
- Los periodistas pueden cambiar la o las palabras a escuchar en cualquier momento.
  - El sistema debe ofrecer una manera de que el usuario introduzca nuevas palabras a escuchar.
  - El sistema debe poder parar los escuchadores activos y tiene que poder iniciarlos de nuevo con las nuevas palabras a escuchar.
  - El sistema debe realizar todas esta tareas sin afectar la experiencia del usuario.
- Los periodistas ven en tiempo real y sin recargar de la página, las actualizaciones que lleguen de las redes sociales.
  - El sistema tiene que poder establecer una conexión con cada cliente y debe guardar una manera para comunicarse con el mismo.
  - El sistema tiene que utilizar esa manera de comunicación para enviarle los nuevos mensajes al cliente en tiempo real sin necesidad de recargar la página.
  - El cliente tiene que renderizar el HTML que le llegue del servidor sin recargar la página.

2.2. Requisitos de arquitectura lógica

Estos requisitos detallan la arquitectura lógica que se impuso por parte de el Diario de Navarra.

- El servidor tiene que ser desarrollado en el lenguaje de programación Python. Para ello, será implementado mediante el *framework* Twisted. Twisted es un *framework* para el desarrollo de aplicaciones de red orientado a la programación dirigida por eventos y escrito en Python.
- El control de acciones en tiempo real dentro del sistema se realizará mediante Cyclone y Websocket (protocolo que provee canales de comunicación *full-duplex* sobre una única conexión de TCP) . Cyclone es un *framework* de servidor web para Python que implementa la API de Tornado (*framework* web para Python que proporciona un conjunto de librerías de red asíncronas) como un protocolo de Twisted. Cyclone nos permite escribir cómodamente, un servidor consistente en clases que definen el comportamiento respecto a las peticiones entrantes de los clientes, como puede verse en la Fig. 2.1

```
class IndexHandler(cyclone.web.RequestHandler):

    @cyclone.web.authenticated  # triggers authentication
    @defer.inlineCallbacks      # allows for inline callbacks
    def get(self):

        result = yield self.do_download()
        self.write(result)
```

Figura 2.1: Ejemplo de clase para la gestión de mensajes con Cyclone.

- Para obtener una presentación de datos estandarizada se utilizará Embedly, que consigue convertir diferentes formatos de información (*tweets*, *posts*, videos,...) en *tarjetas HTML*. Embedly es un conjunto de herramientas que ayudan a crear mejores experiencias de usuario. Los aspectos a destacar de Embedly son la sencillez de uso y la estandarización que ofrece para un gran número de proveedores (entre los que se encuentran las redes sociales que se pueden escuchar). Los datos a formatear con Embedly se obtendrán de diversos medios sociales. La idea es ofrecer distintos tipos de datos o de multimedia correspondientes a la tendencia de la red social (fotos en Flickr, videos en Youtube, texto en Facebook...), pero presentarlos de manera homogénea.

He realizado un diagrama de componentes general (Fig. 2.2) que muestra cómo se divide el sistema completo en componentes, incluyendo el apartado de búsqueda semántica, la aplicación móvil y la API necesaria para integrar los tres módulos.

En la Fig. 2.2 se aprecia como existe un componente intermedio que se encarga de unificar ambos buscadores y ofrece un interfaz que es usado por la API para obtener las tarjetas HTML, enviarlas a los usuarios finales e introducirlas en la base de datos.

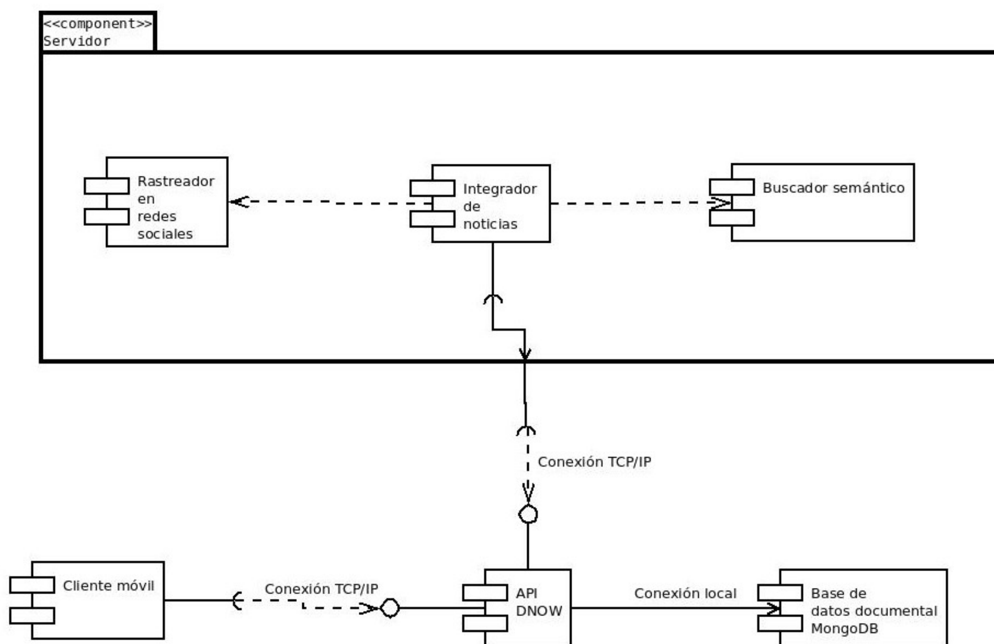

Figura 2.2: Diagrama de componentes general del sistema.

2.3. Requisitos de interfaz gráfica

El Diario de Navarra quiere una interfaz gráfica amigable y optimizada para ser utilizada en un navegador web, es decir, una página web. Para el frontal del periodista, se propone la interfaz que se muestra en la Fig. 2.3, donde vemos una captura obtenida de un prototipo básico del que disponían con anterioridad en el Diario de Navarra. Tal y como se observa en esa figura, la aplicación *backend* dispondrá de tres columnas. En la columna izquierda han de aparecer en orden de llegada (de arriba abajo) todas las tarjetas HTML renderizadas con Embedly. La aplicación frontal del periodista permitirá arrastrar las tarjetas HTML hasta una columna central, desde la cual se enviarán finalmente al cliente real (a aquellos usuarios que se hayan suscrito a la noticia). Por ultimo, en la columna derecha, tenemos el buscador semántico, que si bien actúa de manera análoga a la columna izquierda, queda fuera del alcance de este proyecto.

2.4. Requisitos de arquitectura física

En lo que a requisitos de arquitectura física se refiere, desde el principio se impuso que el periodista fuera capaz de acceder a la aplicación vía navegador web. Esto, por supuesto, implica que se tiene que programar un servidor que responda a las peticiones de un cliente web.

Por otra parte, el servidor debe recibir datos de una serie de escuchadores en redes

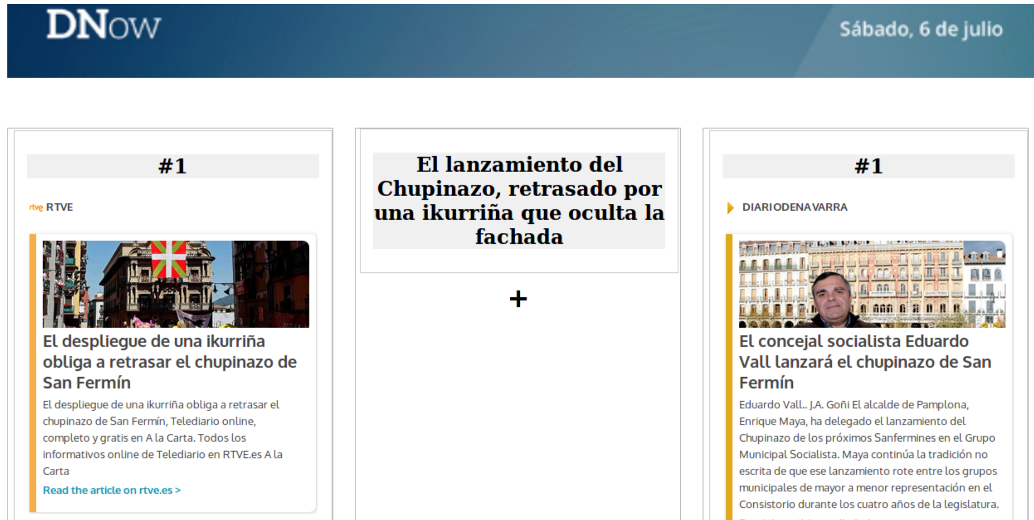

Figura 2.3: Interfáz gráfico propuesto en el prototipo de aplicación.

sociales para que pueda normalizarlos y enviarlos de una manera estándar al cliente.

Capítulo 3

Análisis funcional

3.1. Análisis de casos de uso

Los casos de uso del sistema se recogen en la Fig. 3.1. Tres actores interactúan con el sistema:

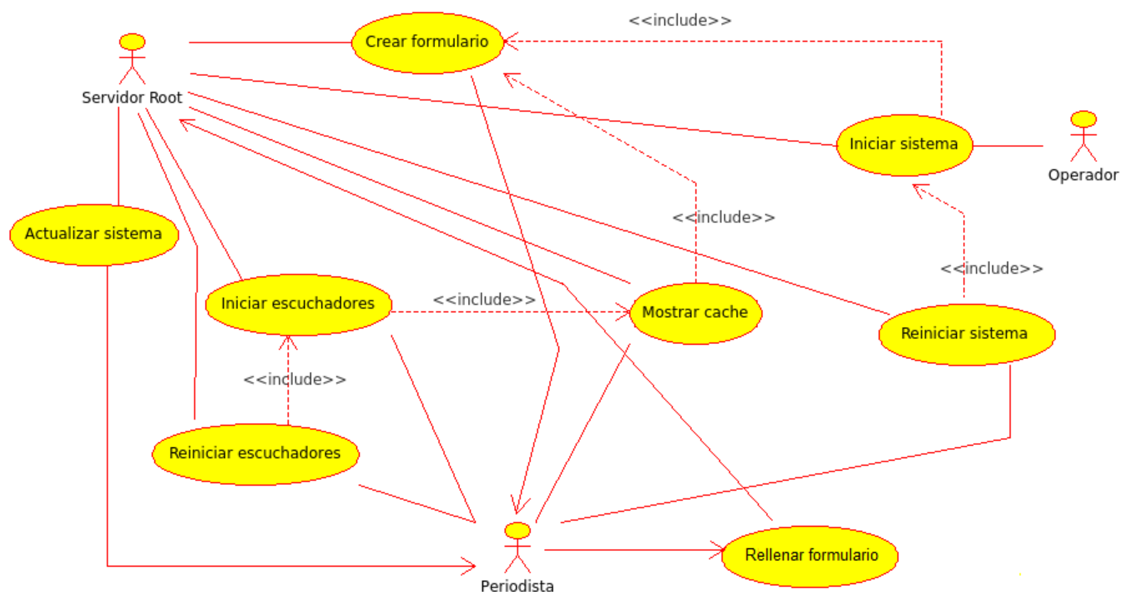

Figura 3.1: Diagrama de casos de uso del sistema

- Operario: Empleado de Diario de Navarra que se encarga de iniciar el sistema.
- Periodista: Empleado de Diario de Navarra que se encarga de filtrar la información que llega de las redes sociales.
- Servidor Root: Actor que representa al servidor. Se encargar de realizar las operaciones que ordenan el periodista y el operario.

A continuación se detalla cada uno de los casos de uso.

Caso de uso: 01- Iniciar sistema (CdU01)

a. *Actores:* Operario y Servidor root.
b. *Descripción:* El operario de Diario de Navarra se ocupa de iniciar el servidor para que los periodistas que vayan a crear hilos puedan iniciar los escuchadores que crean necesarios.
c. *Precondición:* El servidor tiene que estar apagado.
d. *Flujo básico:*
    1. El operario solicita iniciar el servidor;
    2. El servidor se inicia e inicia la caché para almacenar las tarjetas HTML,el manejador de mensajes en tiempo real y el reactor para escuchar las peticiones web de los periodistas;
    3. El sistema se queda en espera de peticiones de los periodistas;
    4. El sistema realiza acciones con los escuchadores dependiendo de la petición que le llegue y de su estado.
e. *Diagrama de actividad:* En el diagrama de actividad de la Fig. 3.2 se muestra como se inicia el sistema y como se comporta dependiendo del tipo de petición que reciba y el estado del mismo. Tras recibir la petición, todas las actividades que aparecen en el diagrama hacen referencia a los diagramas que se detallan más adelante.
f. *Flujo alternativo:* -
g. *Postcondición:* El servidor está iniciado y esperando peticiones. Dependiendo del tipo de petición se devolverá el formulario o la plantilla completa para visualizar las tarjetas HTML.
h. *Condiciones de error:* Si alguno de los servicios que necesita el servidor se encuentran inactivos o no disponibles, se deberá informar al operario por pantalla.
i. *Criterio de aceptación:* Se muestran al operario los log de inicio de Cyclone por consola.

Caso de uso: 02- Crear formulario (CdU02)

a. *Actores:* Servidor Root
b. *Descripción:* El periodista encargado de cubrir una noticia de interés quiere iniciar los escuchadores de redes sociales que crea necesarios y ha realizado la primera petición GET. Se le tiene que enviar el formulario inicial.
c. *Precondición:* El servidor ha debido ser iniciado con anterioridad(CdU01) y el usuario tiene que realizar una petición GET al servidor sin que haya ningún escuchador iniciado.

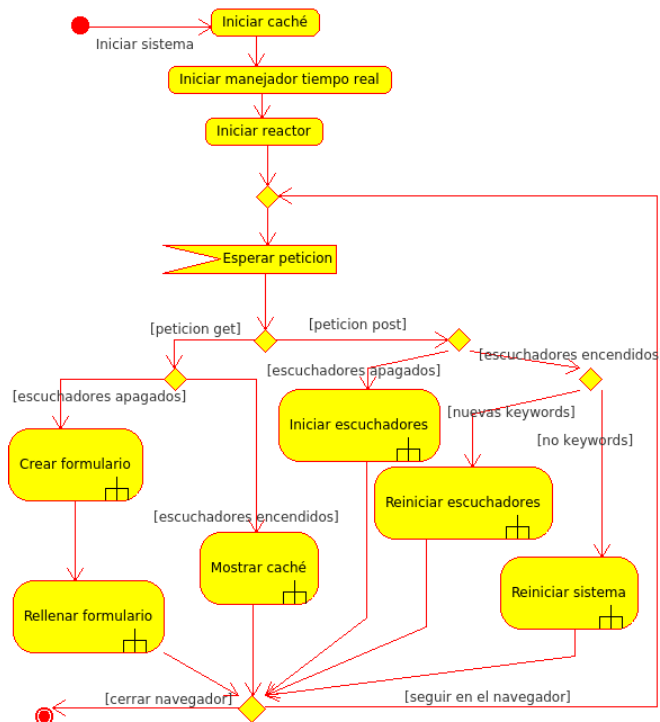

Figura 3.2: Diagrama de actividad para la inicialización del sistema.

d. *Flujo básico:*

   1. El periodista solicita la pagina principal del servidor;
   2. El servidor llama al parseador del archivo de configuración;
   3. El parseador le devuelve que redes sociales pueden ser rastreadas y si tienen opciones;
   4. El servidor construye el formulario, creando para cada red social rastreable un checkbox y un campo oculto de opciones si las tiene;
   5. El servidor envía el formulario al cliente.

e. *Diagrama de actividad:* Para crear el formulario (Fig. 3.3), primero se debe leer el archivo de configuración para determinar que escuchadores tiene disponibles el sistema. Posteriormente se añaden dichos escuchadores y las opciones de cada uno al formulario.

f. *Flujo alternativo:*

   3.a Si el archivo de configuración no contiene ningún escuchador disponible, se devuelve un mensaje de alerta de tipo *No hay escuchadores disponibles*.

g. *Postcondición:* El usuario ve en el navegador un formulario con *checkbox's* que le permiten seleccionar que escuchadores quiere iniciar y bajo que condiciones se ha de buscar si el escuchador tiene opciones.

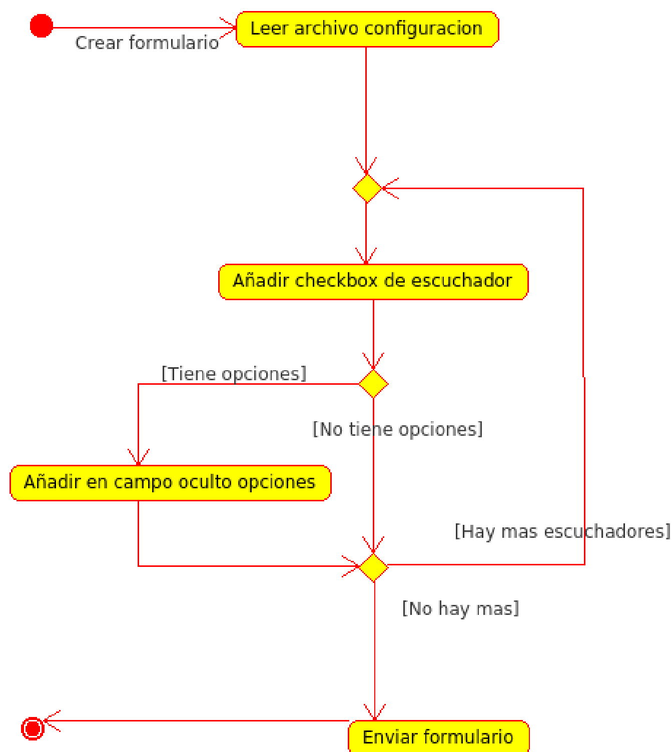

Figura 3.3: Diagrama de actividad para la creación de un formulario.

h. *Condiciones de error:* Si el archivo de configuración no existe se ha de notificar al periodista o al operario para que se cree uno. Si la estructura del archivo de configuración es incorrecta se ha de informar al operario para solventarlo.

i. *Criterio de aceptación:* El usuario a accedido a la pagina principal y le aparece el formulario con los escuchadores disponibles.

Caso de uso: 03- Rellenar el formulario (CdU03)

a. *Actores:* Periodista

b. *Descripción:* El periodista rellena los campos que proporciona el formulario para enviárselo después al servidor.

c. *Precondición:* El servidor se ha iniciado con anterioridad (CdU01) y el servidor le ha servido el formulario al usuario(Cdu02).

d. *Flujo básico:*

   1. El periodista selecciona los escuchadores a activar;
   2. Introduce las palabras a buscar en cada uno;
   3. Introduce las opciones de fecha si el escuchador tiene opciones;

4. Pulsa el botón para enviar el formulario.

e. *Diagrama de actividad:* El periodista elige que rastreadores de los que le aparecen en pantalla quiere activar (Fig. 3.4). Para cada uno de ellos tiene que introducir la o las palabras clave a rastrear y si el rastreador dispone de opciones, ha de decidir si utilizarlas o no.

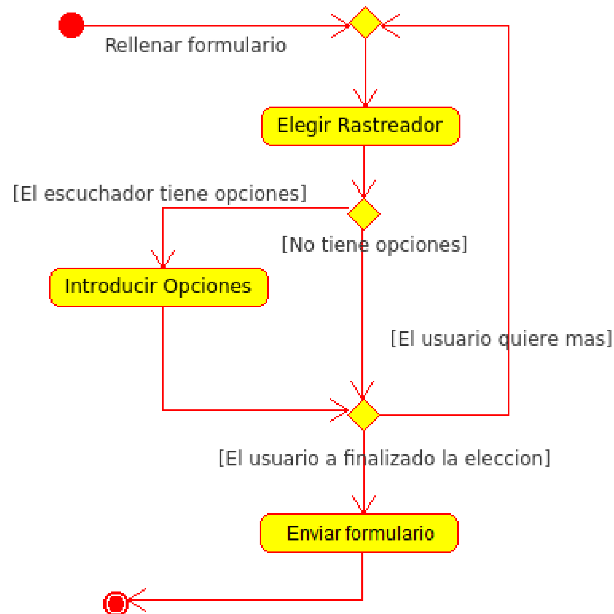

Figura 3.4: Diagrama de actividad para el rellenado de un formulario.

f. *Flujo alternativo:*

2b. Si el periodista accidentalmente no envía ninguna palabra clave, los escuchadores no se inician y el servidor devuelve el formulario de nuevo.

g. *Postcondición:* El usuario recibe la plantilla principal y a medida que lleguen las tarjetas HTML, estas se visualizarán en la plantilla.

h. *Condiciones de error:* Si el periodista no envía ningún dato en el formulario o envía una combinación errónea, no se ha de iniciar ningún escuchador y se devuelve el formulario de nuevo.

i. *Criterio de aceptación:* Los escuchadores que han de escuchar en sus respectivas redes sociales están rastreando y el periodista ve en pantalla las tarjetas HTML a medida que lleguen desde el servidor.

Caso de uso: 04- Iniciar Escuchadores (CdU04)

a. *Actores:* Periodista y Servidor root

b. *Descripción:* El servidor recibe el formulario rellenado anteriormente por el periodista e inicia los escuchadores que haya pedido el periodista.

c. *Precondición:* El servidor ha debido ser iniciado con anterioridad (CdU01) y el periodista ha rellenado el formulario (CdU03).

d. *Flujo básico:*

   1. El servidor obtiene los parámetros provenientes del formulario;
   2. Crea instancias de los escuchadores e hilos para que puedan ejecutarse de manera asíncrona;
   3. Los escuchadores se inician y rastrean las palabras que el periodista ha indicado en el formulario.

e. *Diagrama de actividad:* Dependiendo de como haya rellenado el formulario el periodista, se crearan instancias de los escuchadores así como hilos para que cada escuchador realice su tarea de manera asíncrona (Fig. 3.5).

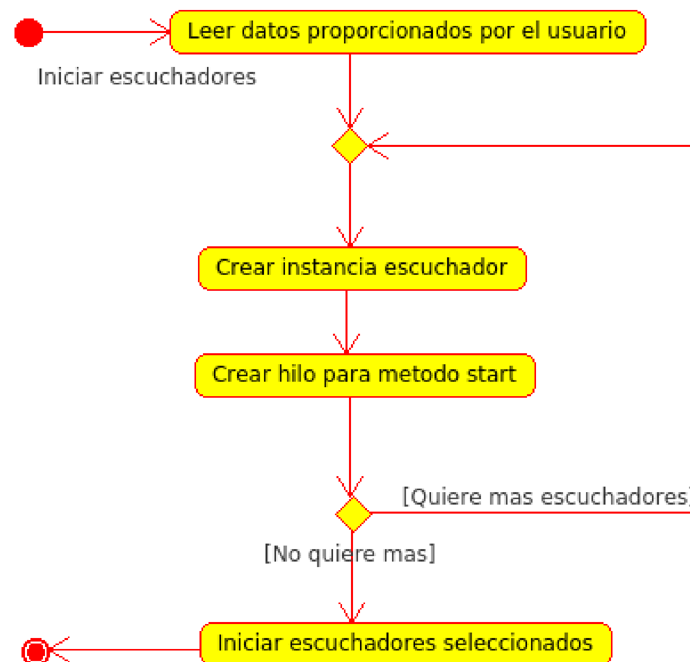

Figura 3.5: Diagrama de actividad para inicializar los escuchadores.

f. *Flujo alternativo:*

   3a. Si un escuchador falla, se para la ejecución de ese escuchador en particular permitiendo al sistema seguir con el resto de escuchadores activos.

g. *Postcondición:* Los escuchadores de redes sociales seleccionados previamente por el periodista han de estar rastreando las redes sociales y se encargan de actualizar la caché.

h. *Condiciones de error:* Si alguna API no está disponible, el servidor tiene que hacerselo saber al periodista.

i. *Criterio de aceptación:* Los escuchadores que haya especificado el periodista al rellenar el formulario rastrean sus respectivas redes sociales y actualizan la caché cuando es necesario.

Caso de uso: 05- Mostrar caché (CdU05)

a. *Actores:* Periodista y Servidor root
b. *Descripción:* Un usuario realiza una petición `GET` con escuchadores rastreando las redes sociales. En esta situación el primer paso es abrir una conexión Websocket y enviarle al usuario la caché para que la renderice.
c. *Precondición:* El servidor tiene que estar activado previamente (Cdu01) y tiene que haber al menos un escuchador activo (Cdu04).
d. *Flujo básico:*
   1. El usuario hace una petición `GET`;
   2. Se abre una conexión Websocket (entre el usuario y el servidor) y se almacena en el servidor para poder enviarle mensajes a ese cliente;
   3. El sistema comprueba que hay escuchadores rastreando;
   4. El sistema devuelve la plantilla predeterminada con las tarjetas HTML que almacena la caché;
   5. En el cliente se realizan las llamadas a Embedly para renderizar las tarjetas HTML.
e. *Diagrama de actividad:* Cuando un usuario se conecta y hay escuchadores activos, primero, se realiza una conexión de Websocket entre el usuario y el servidor. Acto seguido, el servidor envía al cliente la plantilla HTML junto al contenido de la caché. Por ultimo, se renderizan las tarjetas HTML que contenga la caché(Fig. 3.6).
f. *Flujo alternativo:*
   3a. Si no hay escuchadores activos el sistema devolverá el formulario de los escuchadores (CdU02 y CdU03).
g. *Postcondición:* El usuario tiene que ver en pantalla lo que hay en la caché en el momento en el que hace la petición y mediante la conexión Websocket establecida, le han de llegar las nuevas actualizaciones que se envíen a los clientes activos o en caso contrario ve el formulario inicial.
h. *Condiciones de error:* Si el servicio de Embedly está caído, el periodista vera las tarjetas HTML en forma de enlaces a las noticias.
i. *Criterio de aceptación:* El periodista obtiene la plantilla principal con los datos que tenga la caché en ese momento y una conexión de Websocket con el servidor por la que recibe nuevas tarjetas de HTML.

Caso de uso: 06- Actualizar sistema (cache y periodistas en escucha) (CdU06)

a. *Actores:* Periodista y Servidor root.

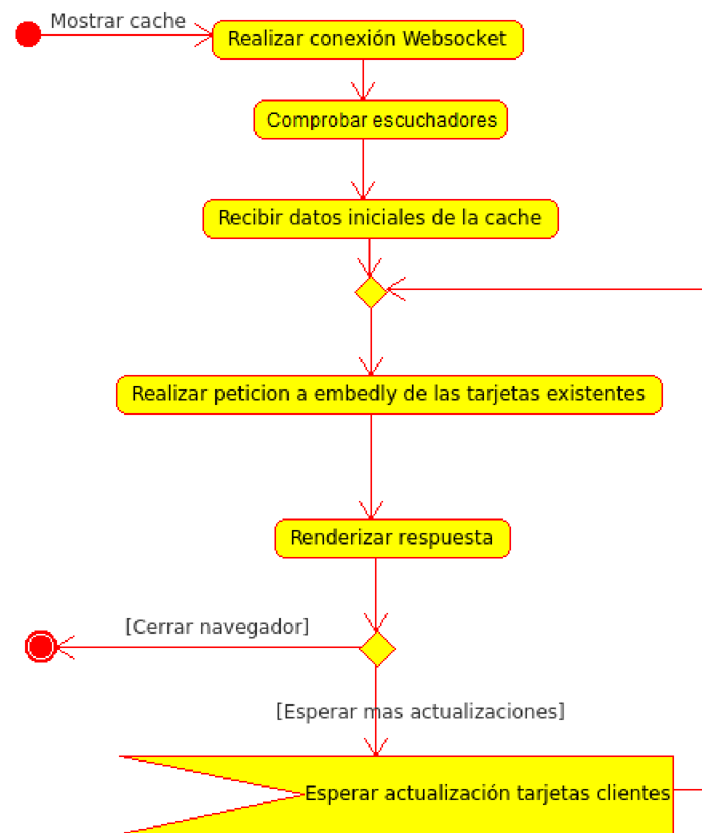

Figura 3.6: Diagrama de actividad para mostrar la caché.

b. *Descripción:* Uno de los escuchadores activos ha recibido un mensaje de su API correspondiente o en caso de los escuchadores estáticos (que cogen todo lo necesario y envían lo recibido eventualmente) a pasado el tiempo establecido para enviar otro mensaje a la caché.

c. *Precondición:* El servidor ha tenido que ser iniciado con anterioridad (CdU01) y debe de haber al menos un escuchador activo (CdU04).

d. *Flujo básico:*

   1. Un escuchador específico recibe datos de parte de la API o se le acaba el tiempo de espera y debe actualizar la caché.
   2. El escuchador construye la tarjeta de HTML necesaria dependiendo de la red social que sea, la introduce en un nodo junto a un *id* y envía el nodo a la caché para actualizarla (en la caché se introduce el HTML unicamente, el id se utiliza en el lado del cliente para evitar duplicados) y enviar las actualizaciones a los clientes;
   3. Se actualiza la caché y se enviá la actualización a los usuarios que estén escuchando;

4. El escuchador en cuestión se queda escuchando más actualizaciones o duerme el tiempo que se le haya estipulado.

e. *Diagrama de actividad:* Para actualizar el sistema, primero, se esperan nuevos datos de las APIs que se estén escuchando. A continuación se construye la tarjeta HTML a partir de los datos obtenidos. Por ultimo, se actualiza la caché (se ha de bloquear el método que actualiza la caché para evitar modificaciones simultaneas) y se envía la actualización a los usuarios activos(Fig. 3.7).

f. *Flujo alternativo:*

   1.a. Si el mensaje recibido no es válido el escuchador lo ignora y sigue esperando nuevos mensajes.

g. *Postcondición:* Si el mensaje era válido, la caché ha de estar actualizada con la tarjeta HTML de el mensaje mencionado anteriormente. En cualquier caso, el escuchador que recibe el mensaje o actualiza la caché ha de seguir corriendo una vez el proceso a terminado.

h. *Condiciones de error:* No recibir datos validos desde la API que envíe la información.

i. *Criterio de aceptación:* El escuchador particular que ha recibido datos sigue escuchando y en caso de que los datos recibidos sean validos, la caché esta actualizada con esos datos en forma de tarjeta HTML.

Caso de uso: 07- Reiniciar escuchadores con nuevas palabras clave (CdU07)

a. *Actores:* Periodista y Servidor root.

b. *Descripción:* El periodista ha decidido cambiar las palabras que se van a escuchar en las redes sociales y envía en tiempo real las nuevas palabras clave a escuchar. El sistema tiene que parar los escuchadores que se encuentren activos, iniciar unos nuevos con las nuevas palabras a escuchar y empezar a actualizar la caché y la columna del periodista sin necesidad de actualizar la página.

c. *Precondición:* El servidor tiene que estar activado previamente (CdU01), el periodista ha rellenado el formulario (CdU03) y tiene que haber al menos un escuchador activo (CdU04).

d. *Flujo básico:*

   1. El usuario introduce nuevas palabras a buscar en los campos que se proporcionan en la plantilla;
   2. Las nuevas palabras clave se transmiten al servidor en tiempo real para que este realice las acciones pertinentes;
   3. El servidor obtiene las palabras y para todos los escuchadores de redes sociales que estén activos en ese momento;
   4. El servidor inicia de nuevo los escuchadores con las nuevas palabras que le han llegado desde el cliente.

e. *Diagrama de actividad:* Si un periodista rellena de nuevo el formulario y lo envía, se reiniciarán los escuchadores dependiendo de los parámetros que envíe. Para ello, en primer lugar, se pararán los escuchadores que estén activos. Después se reiniciaran los escuchadores en base a los datos que haya introducido el periodista en el formulario(Fig. 3.8).

f. *Flujo alternativo:*

   1.a Si el mensaje recibido por parte del periodista no contiene palabras, la ejecución de los escuchadores activos no ha de detenerse.

g. *Postcondición:* Tiene que haber escuchadores activos dependiendo de los nuevos datos que haya introducido el periodista en el formulario.

h. *Condiciones de error:* El usuario pulsa el botón de enviar sin introducir ninguna palabra previamente.

i. *Criterio de aceptación:* El periodista observa, sin necesidad de recargar la página, la llegada de tarjetas HTML relacionadas con las nuevas palabras que haya introducido a rastrear.

Caso de uso: 08- Reiniciar sistema (CdU08)

a. *Actores:* Periodista y Servidor root

b. *Descripción:* El periodista quiere borrar todo el contenido actual de la caché y empezar el rastreo desde cero.

c. *Precondición:* El servidor tiene que estar activado previamente (CdU01) y tiene que haber al menos un escuchador activo (CdU04).

d. *Flujo básico:*

   1. El periodista pulsa el botón;
   2. Se paran todos los escuchadores activos;
   3. Se inicializan todos los parámetros de la caché a sus valores por defecto;
   4. Se inicia el sistema;
   5. Se redirige al usuario a la pantalla de inicio.

e. *Diagrama de actividad:* Un periodista puede elegir reiniciar el sistema entero. Para ello, ha de pulsar el botón que se proporciona. Al pulsar el botón se pararán los escuchadores, se inicializara todo el sistema y se redirigirá al periodista a la pantalla de inicio(Fig. 3.9).

f. *Flujo alternativo:* -

g. *Postcondición:* El usuario ve en el navegador un formulario con checklists que le permiten seleccionar que escuchadores quiere seleccionar y las opciones que tienen.

h. *Condiciones de error:* -

i. *Criterio de aceptación:* El usuario ve en pantalla la interfaz inicial con el formulario para elegir escuchadores.

3.2. Visión dinámica del sistema

El comportamiento dinámico del sistema queda especificado por los siguientes diagramas de actividad que se han especificado en los casos de uso:

- Iniciación del sistema (Fig. 3.2);
- Crear un formulario (Fig. 3.3);
- Rellenar un formulario (Fig. 3.4);
- Inicializar escuchadores (Fig. 3.5);
- Mostrar caché (Fig. 3.6);
- Actualizar el sistema (Fig. 3.7),
- Reiniciar los escuchadores (Fig. 3.8),
- Reiniciar sistema (Fig. 3.9).

3.3. Diagrama de secuencia del cliente

He realizado un diagrama de secuencia genérico para describir la comunicación a alto nivel entre el cliente y el servidor. Dicho diagrama se encuentra en la Fig. 3.10.

En este diagrama, podemos observar que para empezar, contamos con la precondición de que el servidor haya sido iniciado. Después, el periodista realiza una petición `GET` al servidor y se produce el apretón de manos típico de HTTP.

El servidor comprueba el estado de los escuchadores y si no hay ninguno activo, pregunta al *parser* que escuchadores hay disponibles, construye el formulario y se lo envía al periodista. Este último rellena el formulario y lo envía de vuelta al servidor. Si hay escuchadores activos, se envía la plantilla y la caché. Después de cualquiera de los dos casos, se realiza un bucle en el que se renderizan las tarjetas HTML que no hayan sido renderizadas y se esperan las nuevas tarjetas HTML que envía el servidor.

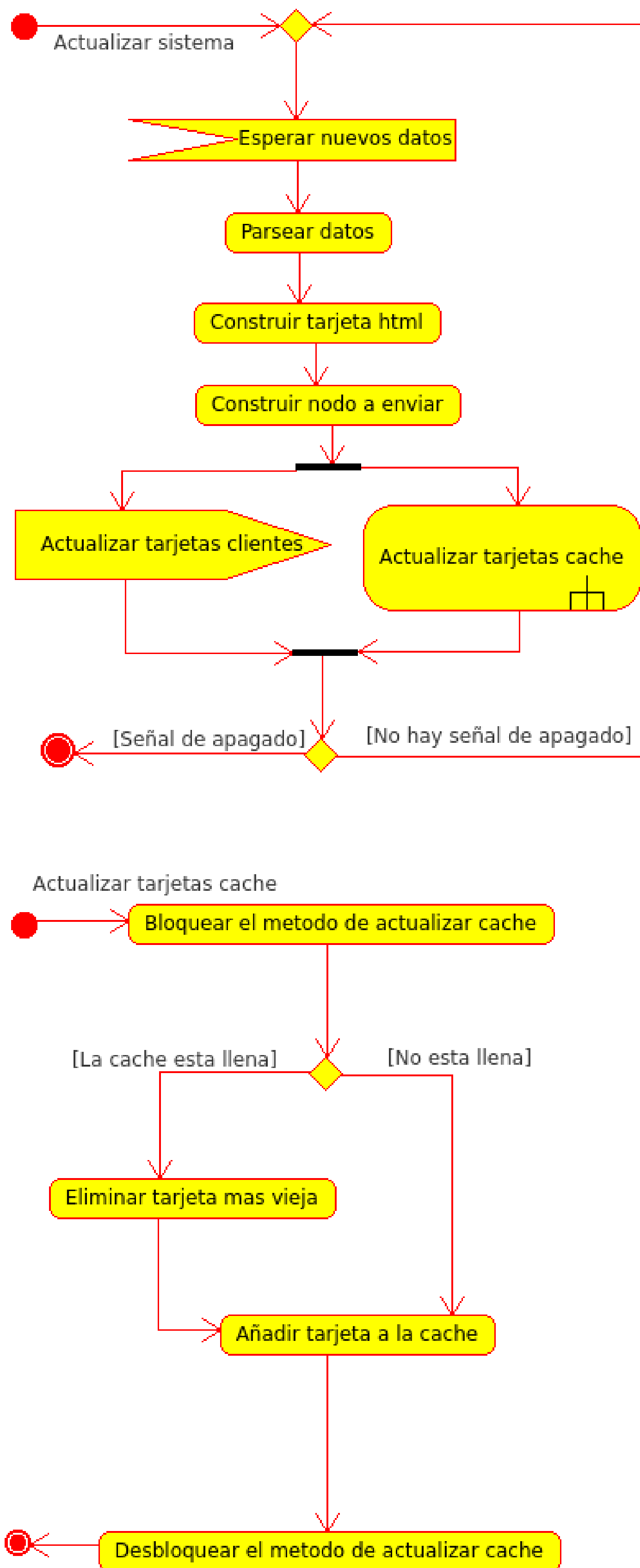

Figura 3.7: Diagrama de actividad para la actualización del sistema.

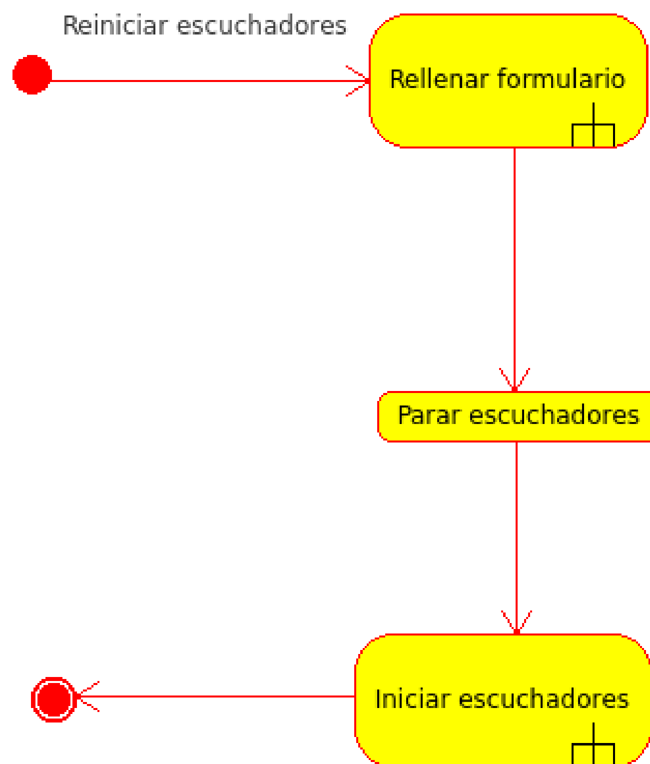

Figura 3.8: Diagrama de actividad la reinicialización de los escuchadores.

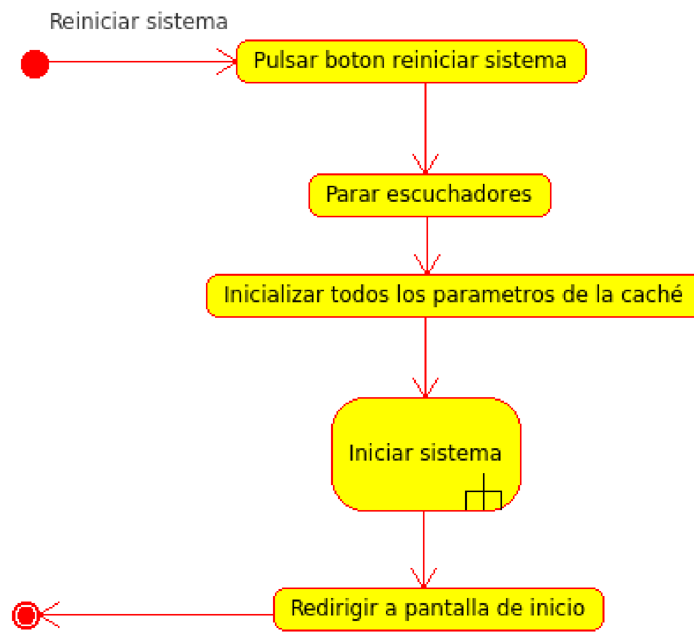

Figura 3.9: Diagrama de actividad la reinicialización del sistema.

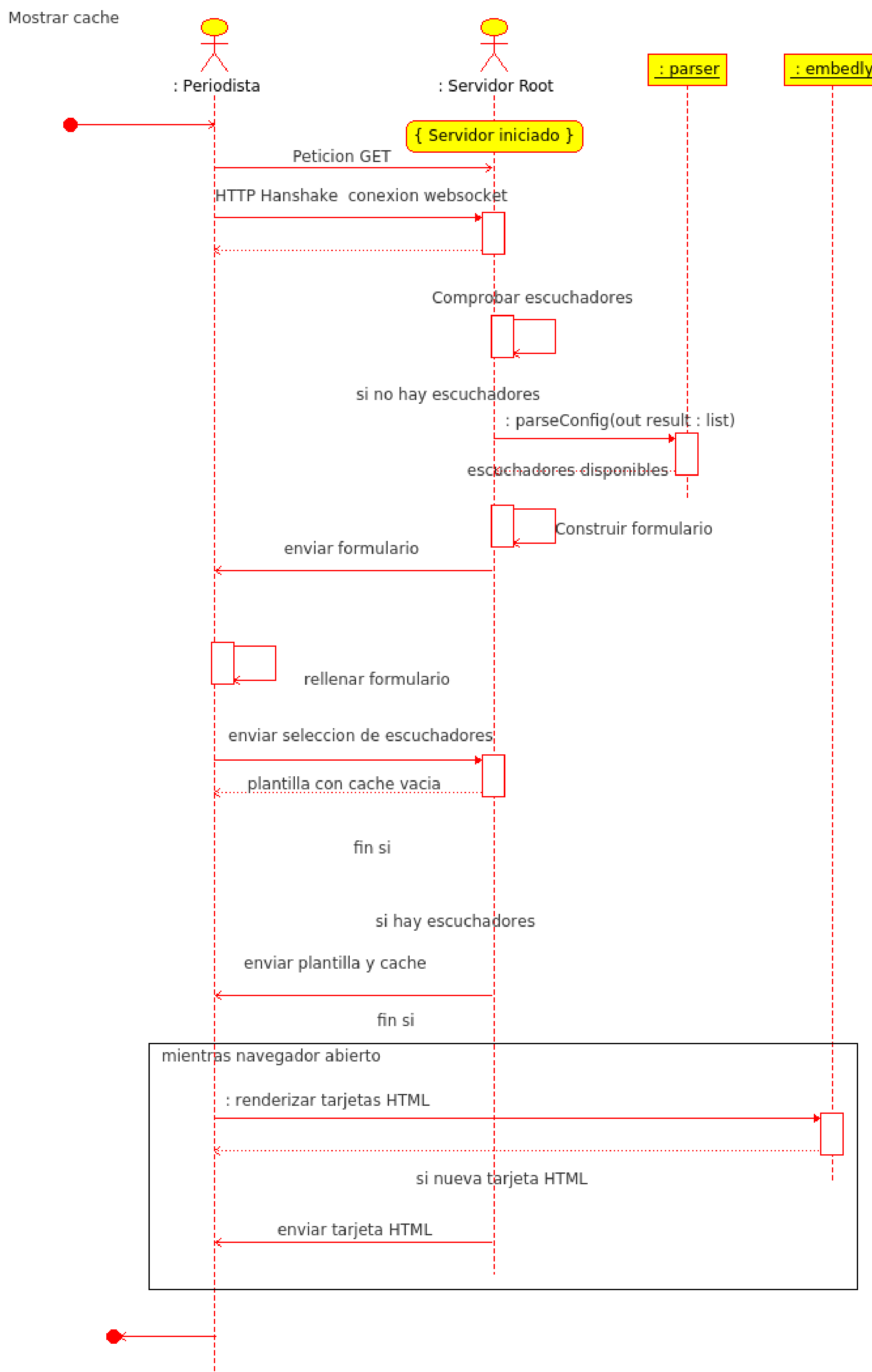

Figura 3.10: Diagrama de secuencia del cliente.

Capítulo 4

Diseño técnico

4.1. Componentes del sistema

El diagrama de componentes general del sistema describe como se divide el proyecto DNow en componentes y se encuentra en la Fig. 2.2 en el apartado de requisitos.

En el diagrama de componentes específico (Fig. 4.1) se muestran los componentes asociados al módulo de búsqueda en redes sociales a un nivel más profundo.

Tal y como se aprecia en la Fig. 4.1, se disponen de tres componentes:

- **Componente servidor**: Este componente se encarga de varias funcionalidades del sistema. Las más importantes incluyen el responder a las peticiones de los clientes y atender a los cambios que se produzcan en la redes sociales. Cuando un escuchador reciba datos, los enviará al servidor mediante el interfaz que este proporciona y el servidor se encargará de enviar los datos al cliente.

  Las actividades que realiza este componente se pueden apreciar en la Fig. 4.2. Para empezar, el servidor se ha de iniciar. Acto seguido, entra en un bucle en el que al mismo tiempo realiza dos acciones, manejar los datos de los escuchadores y manejar las peticiones web.

  En todo momento, el servidor espera datos de parte de los escuchadores para actualizar la cache y enviar los datos a los clientes que estén activos.

  Por otra parte y al mismo tiempo, espera peticiones web desde el cliente. Si un cliente realiza una petición de tipo `POST` y no hay ningún escuchador activo, significa que hay que activar escuchadores según lo que indiquen los parámetros. En caso contrario, se deberán reiniciar los escuchadores con los parámetros recibidos del cliente. Si el usuario realiza una petición `GET` es que desea ver el contenido de la cache.

- **Componente escuchadores**: Este componente espera notificaciones de las API's a las que haya solicitado datos. En cuanto recibe datos, se formatean a un HTML estándar y se envían al servidor.

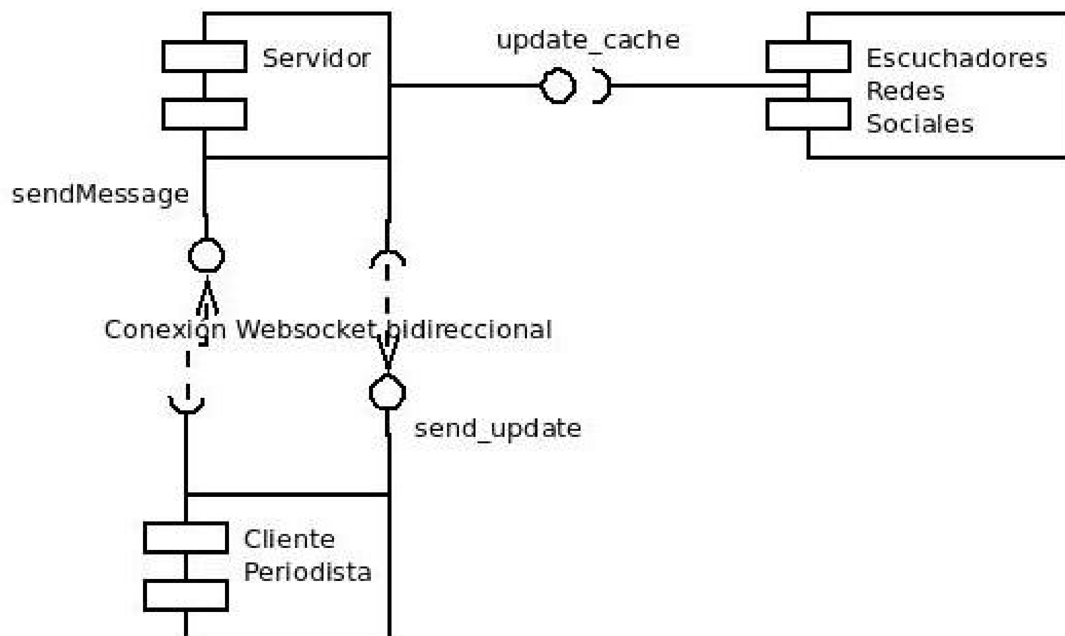

Figura 4.1: Diagrama de componentes específico del sistema.

Las actividades relacionadas con este componente se detallan en la Fig. 4.3. Primero, debe esperar que el servidor le mande la señal de inicio. Acto seguido, analizará los parámetros que le haya enviado el servidor e iniciará el rastreo en las redes sociales. Después entrará en un bucle en el que, primero, esperará datos provenientes de las API's, estandarizará y formateará esos datos a HTML y por último, enviará esos datos estandarizados al servidor.

Al mismo tiempo, esperará la señal para reiniciar los escuchadores. Si la recibe, apagará los escuchadores y acto seguido iniciará el rastreo con lo que le haya enviado el servidor.

- **Componente cliente**: El componente del cliente establece una conexión con el servidor y puede realizar varias operaciones. Entre ellas, puede mandar iniciar o reiniciar los escuchadores, solicitar los datos en la cache y esperar datos en tiempo real desde el servidor para renderizarlos.

  Sus actividades quedan reflejadas en la Fig. 4.4. El primer paso consiste en es-

tablecer una conexión con el servidor. Después, el cliente, deberá seleccionar los escuchadores a activar y dar la orden de encender los mismos pulsando el boton *Start Tracking!*. Acto seguido, entrará en un bucle en el que decidirá si quiere esperar datos o reiniciar los escuchadores con nuevas palabras clave. Si se queda esperando datos del servidor, cuando los reciba, se encargará de renderizarlos para mostrarlos al periodista. En cambio, si el periodista elige reiniciar los escuchadores, deberá de rellenar un formulario y enviarlo al servidor.

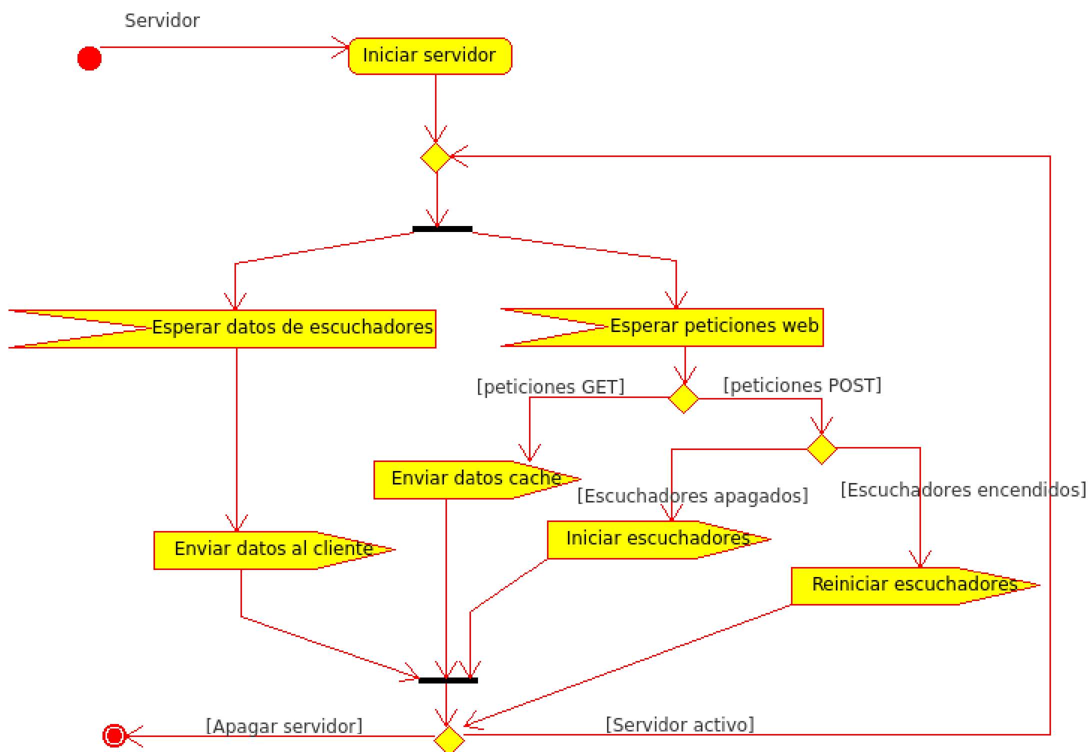

Figura 4.2: Diagrama de actividad del componente servidor

4.2. Interfaces del sistema

El sistema dispone de dos interfaces generales en los que el periodista realiza las acciones de iniciar la búsqueda y reiniciar la búsqueda. En ambas interfaces, la pantalla queda divida en dos columnas. La primera está reservada para las tarjetas HTML o para un espacio vacío si se trata de la interfaz de inicio y la segunda columna (a la derecha) está reservada para el formulario de los escuchadores. En la Fig. 4.5 podemos observar el sistema en un estado inicial. El sistema permite al periodista introducir que escuchadores quiere activar seleccionando el *checkbox* asociado e introduciendo que

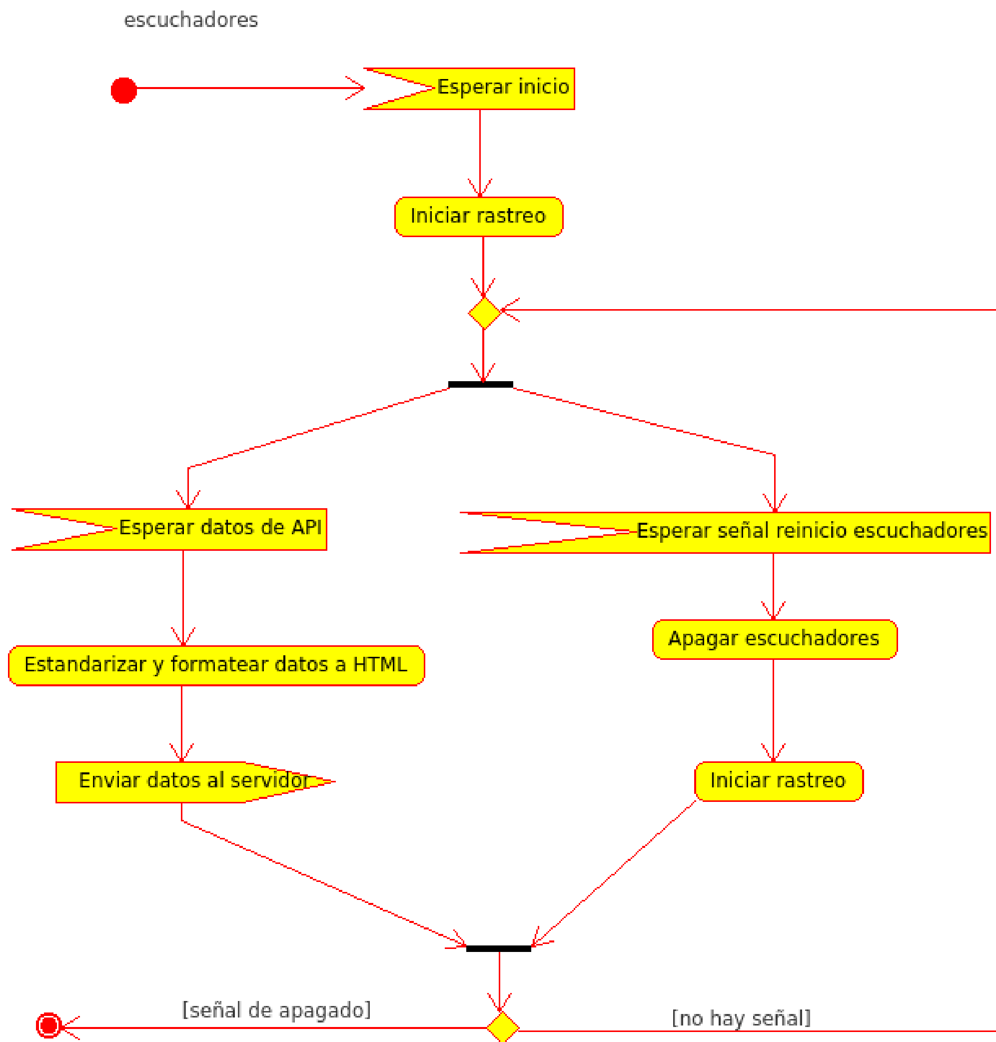

Figura 4.3: Diagrama de actividad del componente escuchadores

palabras desea buscar en cada escuchador como se muestra en la Fig. 4.6. Acto seguido, tiene que presionar el botón "Start Tracking!"para iniciar los escuchadores.

Una vez estén los escuchadores iniciados, el periodista verá una pantalla similar a la Fig. 4.7 (hasta que le lleguen noticias la columna izquierda estará vacía). En esta pantalla, la división en columnas de la pantalla se hace más notoria en cuanto empiezan a llegar tarjetas HTML. Además, en esta interfaz, la columna derecha (el formulario) esta fijada a la pantalla y aunque bajemos con el scroll para ver más tarjetas HTML, el formulario seguirá fijo. De esta manera, se crea un experiencia de usuario más rica y sin necesidad de subir al tope de la página para reiniciar los escuchadores. Como se puede observar, en esta pantalla podemos rellenar de nuevo el formulario para introducir nuevas palabras

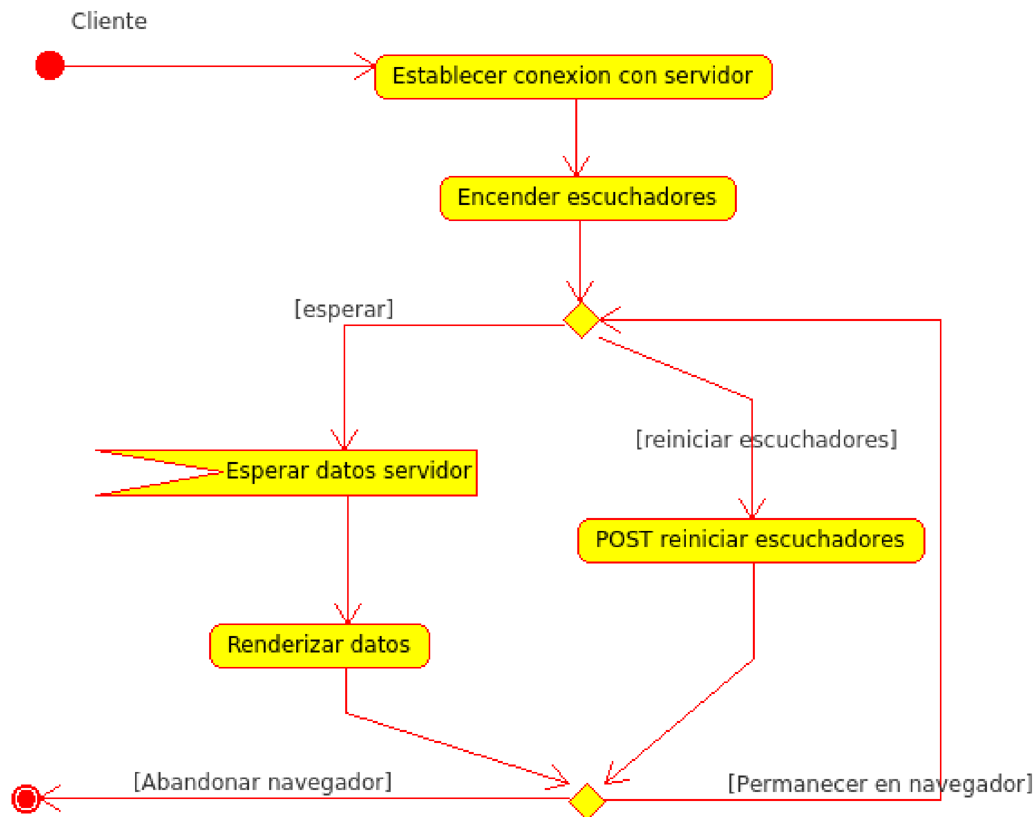

Figura 4.4: Diagrama de actividad del componente cliente

a rastrear o podemos pulsar "Restart the entire system". Si el periodista pulsa este botón, se inicializara todo el sistema (volviendo a los valores por defecto) y se redirigirá a la pantalla de inicio.

4.3. Diagrama de clases

Los diagramas de clases del sistema se recogen en la Fig. 4.8 y la Fig. 4.9. Se puede apreciar como todas las clases que he desarrollado (exceptuando el parser) se relacionan con la clase de la caché (embedCache en el componente servidor). Esto es debido a que cada clase tiene la instancia de la caché para poder actualizarla cuando sea necesario. He realizado los diagramas de clases incluyendo los componentes para mostrar visualmente que clases forman dichos componentes. Como se aprecia en la Fig. 4.8, el componente de los escuchadores esta formado por las clases de todos los escuchadores disponibles y sus dependencias. El componente del servidor(Fig. 4.9) esta formado por la clase `embedCache` (realmente, además de ser la caché de las tarjetas HTML, es el controlador principal del sistema), por la clase `MainHandler` (que se encarga de atender las peticiones web), la clase `RealTimeHandler` (que se encarga de gestionar las acciones

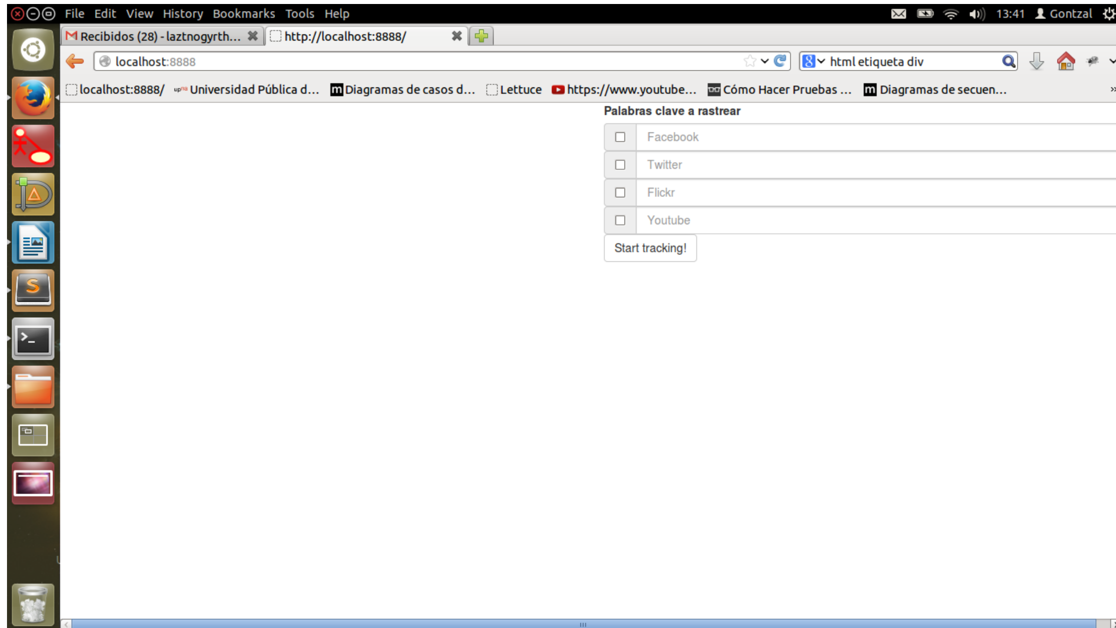

Figura 4.5: Interfaz de inicio.

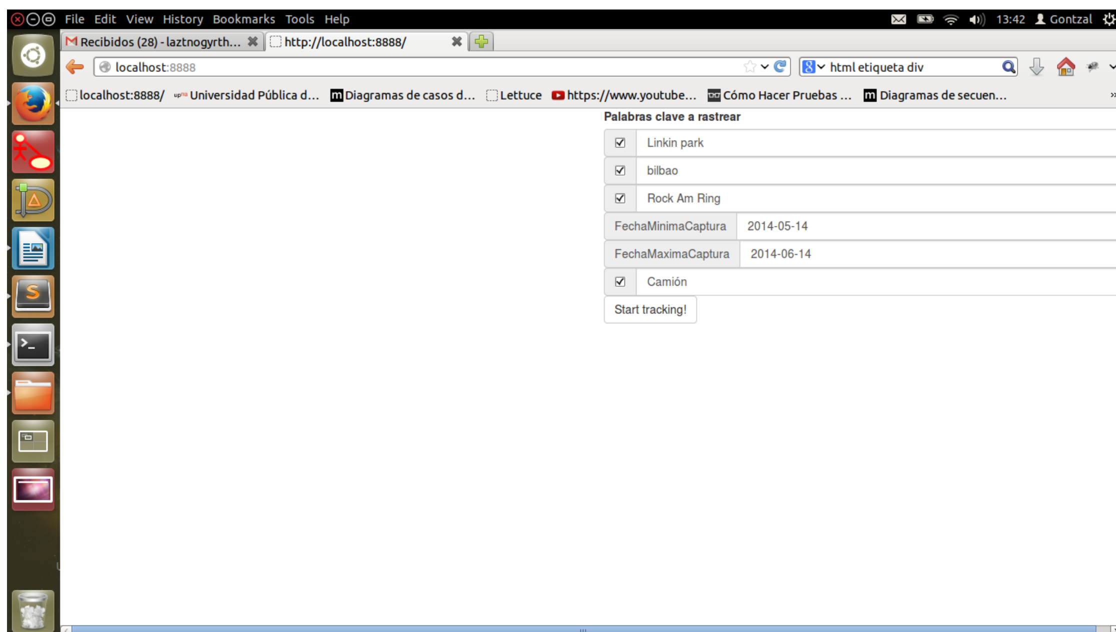

Figura 4.6: Interfaz de inicio en el proceso de introducir datos.

en tiempo real), la clase `ShutDown` (maneja el reinicio completo del sistema) y el *parser* (que se encarga de leer el archivo de configuración y pasar al servidor una lista con los escuchadores disponibles).

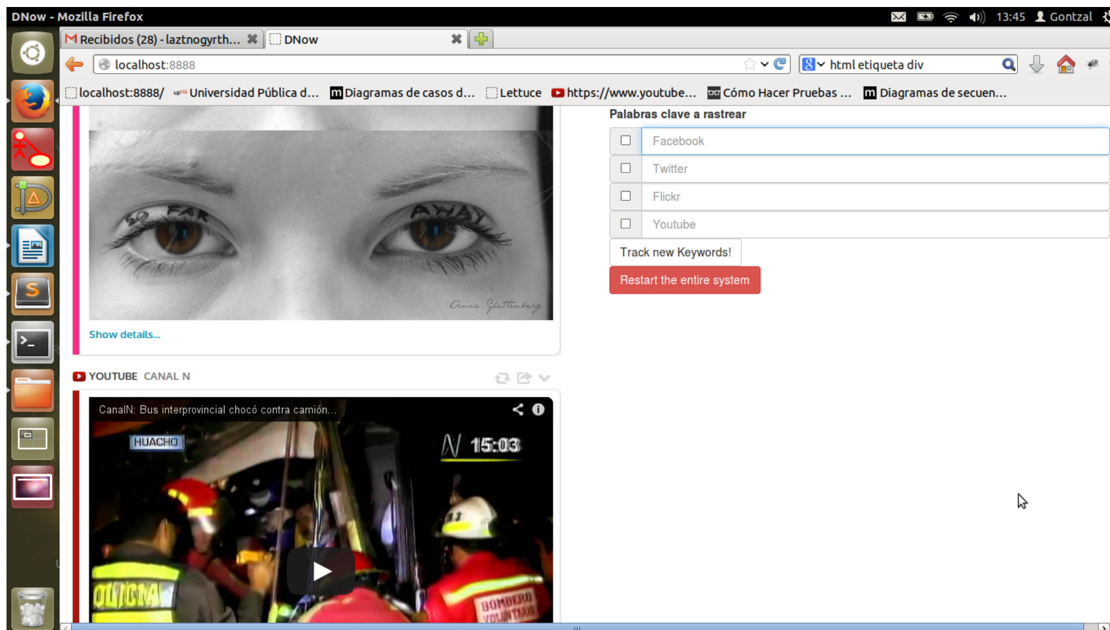

Figura 4.7: Interfaz con escuchadores activos.

4.4. Protocolos de comunicación

Dentro del sistema, la comunicación en tiempo-real entre el cliente y el servidor se maneja usando WebSocket, un protocolo que viene definido en el RFC 6455. El protocolo WebSocket habilita un canal de comunicación bidireccional entre un cliente, que corre código no fiable en un entorno controlado, y un host remoto que responde a las peticiones de ese código. El modelo de seguridad que usan estas comunicaciones es *origin-based security model*, el más común en los navegadores actuales.

El protocolo consiste en una negociación de apertura (*HTTP handshake*) , seguido de mensajes servidos sobre TCP. El objetivo de esta tecnología es proporcionar un mecanismo de comunicación bidireccional para las aplicaciones web que necesitan comunicarse con servidores que no confían en abrir múltiples conexiones HTTP.

En la Fig. 4.10 podemos ver un ejemplo de implementación de un servidor *echo* sobre Websocket. En el, se puede observar que cuando el servidor recibe un mensaje, responde al cliente un mensaje con el contenido que ha recibido.

El protocolo WebSocket no es una conexión HTTP estándar. Después de negociación de apertura, que sí es HTTP, el protocolo está basado en mensajes. Por ello, la mayoría de facilidades HTTP de las que dispone Cyclone no están disponibles en controladores de este tipo. El único método de comunicación disponible en este aspecto es `sendMessage()`. Si el código que vemos en la Fig. 4.10 lo asociamos (en Cyclone) a la ruta `http://localhost:8888/websocket`, y creamos un Javascript como el de la Fig. 4.11, al acceder a la página, obtendremos en una alerta la respuesta que dirá «`You said: Hello,world`».

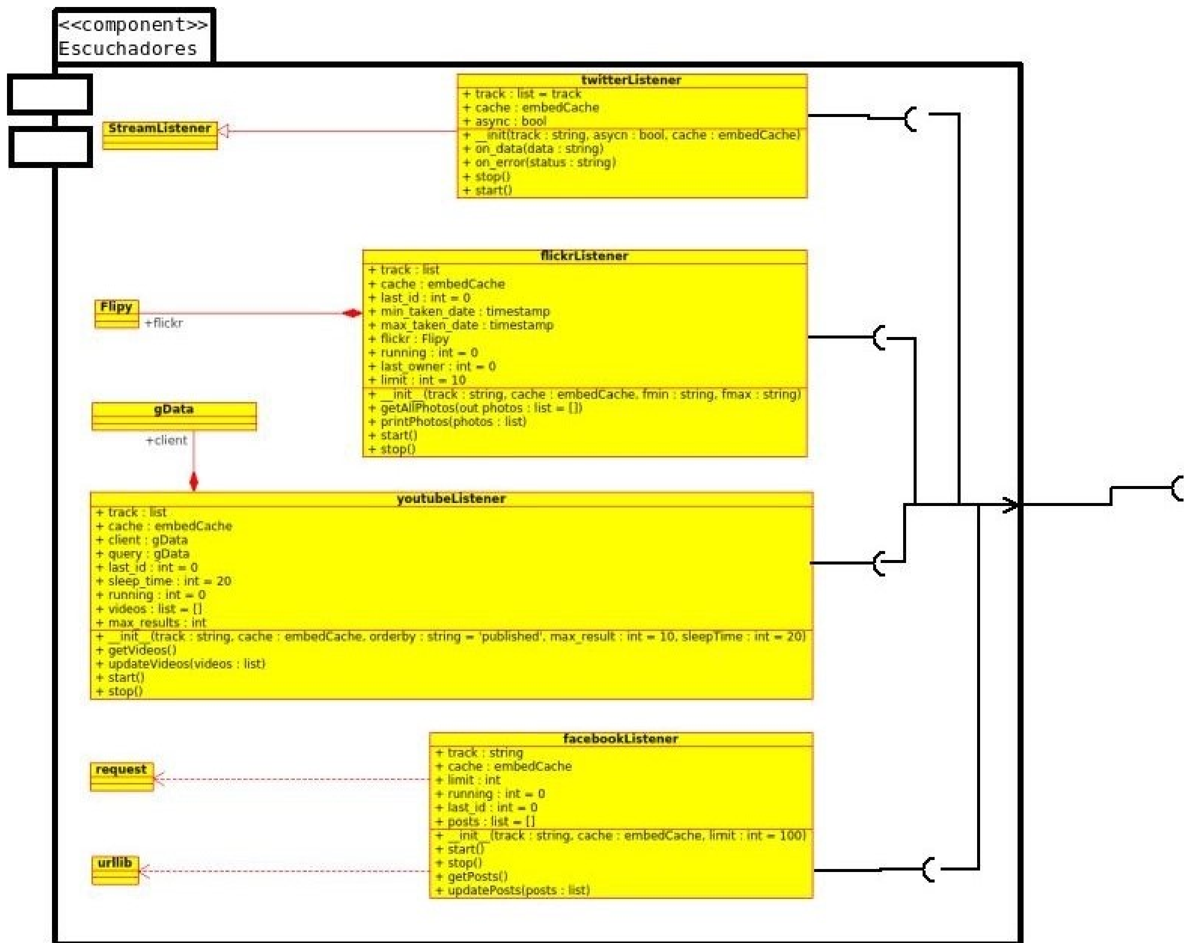

Figura 4.8: Diagrama de clases del componente escuchadores.

4.5. Formatos de intercambio

El sistema utiliza un archivo de configuración codificado en XML en el que se especifica que escuchadores tiene disponibles. De esta manera y siempre que tengamos el escuchador implementado, es posible añadirlo al sistema en cualquier momento incluso teniendo el sistema iniciado. El archivo de configuración tiene una estructura como la que se muestra en la Fig. 4.12.

Se puede apreciar como el archivo define dos marcas generales “Config” y “SocialNetworks” en la que se indica la versión en la que se encuentra el sistema. Dentro de “SocialNetworks” es donde se introducen los escuchadores que deseamos que estén disponibles en forma de marcas. Si no queremos que se pueda elegir ninguna opción a la hora de iniciar el escuchador, introduciremos unicamente como marca el nombre del escuchador (por ejemplo <Twitter></Twitter>). En caso contrario, tenemos que introdu-

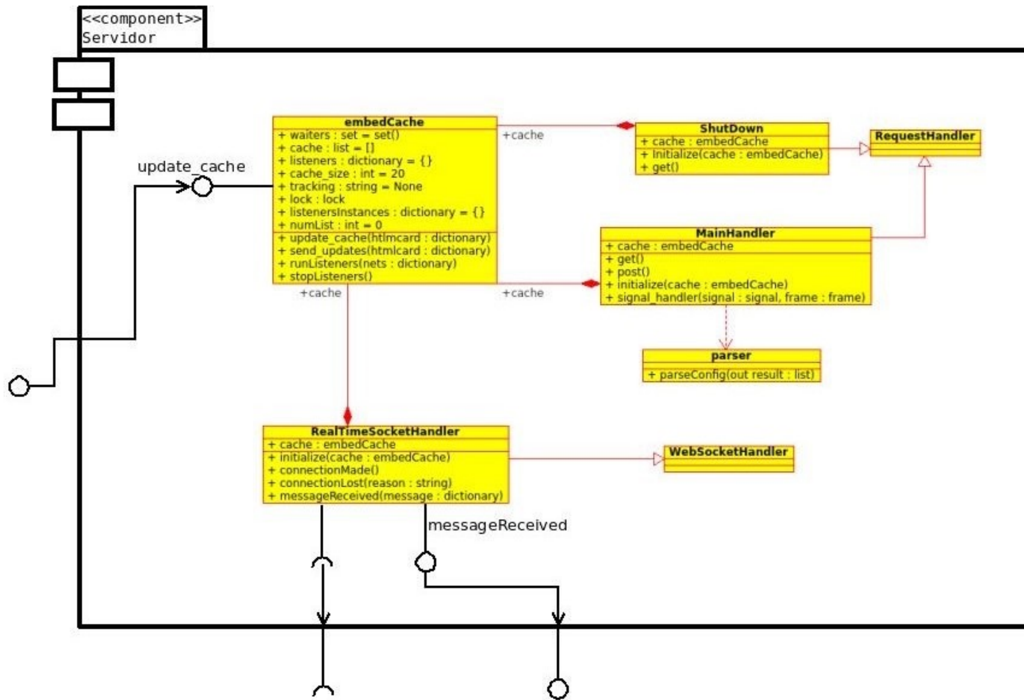

Figura 4.9: Diagrama de clases del componente servidor.

```
class EchoWebSocket(websocket.WebSocketHandler):
        def connectionMade(self):
                print "WebSocket connected"

        def messageReceived(self, message):
                self.sendMessage(u"You said: " + message)

        def connectionLost(self, reason):
                print "WebSocket disconnected"
```

Figura 4.10: Servidor de *echo* implementado sobre WebSocket.

```
var ws = new WebSocket("ws://localhost:8888/websocket");
ws.onopen = function() {
   ws.send("Hello, world");
};
ws.onmessage = function (evt) {
   alert(evt.data);
};
```

Figura 4.11: Código JavaScript de cliente WebSocket

cir en la marca del escuchador las marcas que definan las opciones que puede introducir el periodista a la hora de iniciar el escuchador. Como ejemplo, en la Fig. 4.12 podemos observar como el escuchador de Flickr tiene definidas dos opciones, FechaMinimaCap-

```
<Config>
    <SocialNetworks version="0.1">
        <Twitter>
        </Twitter>
        <Flickr>
            <FechaMinimaCaptura key='FechaMinimaCaptura'/>
            <FechaMaximaCaptura key='FechaMaximaCaptura'/>
        </Flickr>
        <Youtube>
        </Youtube>
        <Facebook>
        </Facebook>
    </SocialNetworks>
</Config>
```

Figura 4.12: Contenido del archivo de configuración.

tura y FechaMaximaCaptura. Hacen referencia a la fecha mínima y a la fecha máxima de captura en las que han debido ser capturadas las imágenes para que el escuchador de Flickr las entregue al servidor.

Para obtener los datos del archivo de configuración existe la clase *"parser"*. Este *parser* lo único que hace es leer el archivo de configuración, y para cada red social que encuentra, determina si tiene opciones y si las tiene, introduce las opciones en una lista de Python que luego introduce en un diccionario como en el siguiente ejemplo:

```
opciones=[opcionesDeRedSocial]
dic[nombreredsocial]=opciones
```

En caso contrario, el diccionario final unicamente tendrá la clave con el nombre de la red social y la lista de opciones será una lista vacía.

4.6. Diagrama de despliegue

El diagrama de despliegue del sistema se recoge en la Fig. 4.13. En el se puede observar una idea general de como se debería desplegar el módulo de búsqueda en redes sociales. Se necesita un servidor para los componentes de servidor y escuchadores. Cada cliente necesitara un navegador web para poder visualizar la página y se abrirá una conexión Websocket entre el servidor y el cliente para el envío de mensajes.

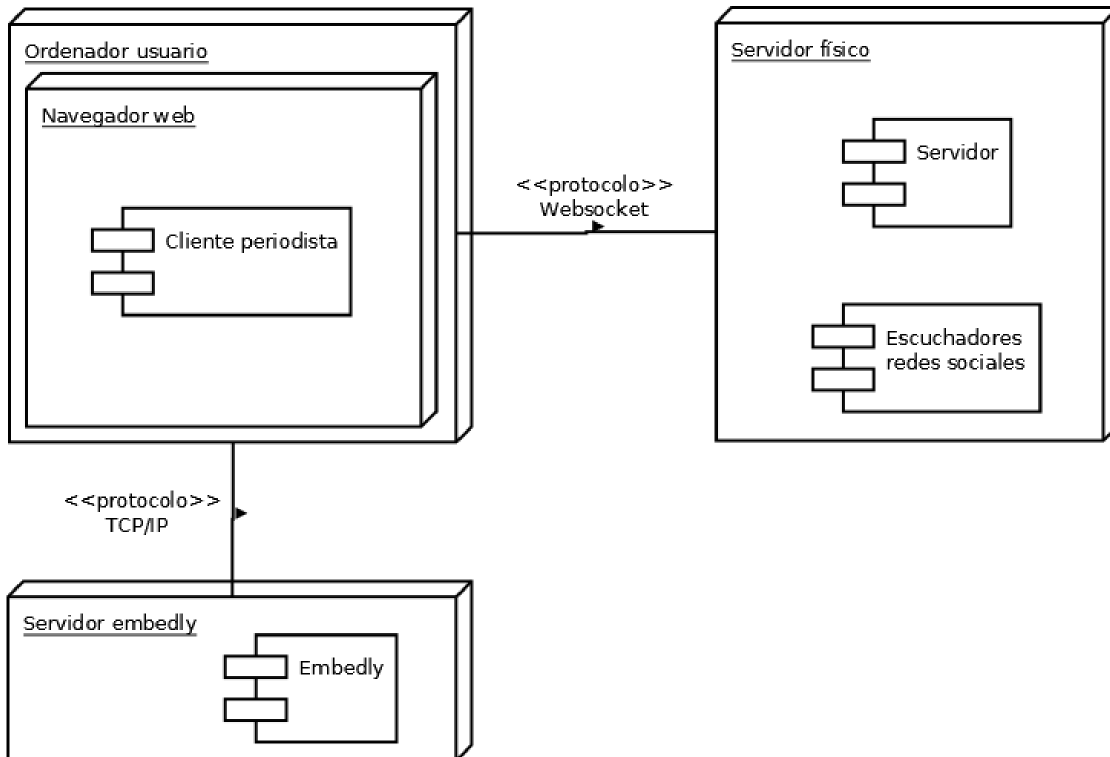

Figura 4.13: Diagrama de despliegue del sistema.

Capítulo 5

Uso del sistema

En este capítulo se incluyen la guía de instalación (para sistemas basados en Debian) y dos diferentes guías de usuario dependiendo del rol ejercitado (operario o periodista).

5.1. Guía de instalación

El proceso de instalación del sistema se compone de tres pasos:

1. Navegar mediante el comando `cd` de la consola a la ubicación actual de la aplicación.
2. Instalar `pip` y `virtualenv`, si no están ya instalados en el sistema. Esto se puede hacer ejecutando los siguientes comandos:

   ```
   sudo apt-get install python-pip python-dev build-essential
   sudo pip install -upgrade pip
   sudo pip install -upgrade virtualenv
   ```

   En caso de éxito, la pantalla mostrada debería parecerse a la que se encuentra en la Fig. 5.1.
3. Ejecutar en consola `sudo pip install -r requiriments.txt` y esperar a que se instalen las dependencias. De nuevo, el resultado debería ser parecido al mostrado en la Fig. 5.2.

5.2. Guía de uso básico del servidor

La prueba del servidor consta de tres pasos, a los que se añade una cuarta fase de comprobación del resultado de la misma. El proceso es tal y como sigue:

```
install --upgrade virtualenv
Downloading/unpacking virtualenv
  Downloading virtualenv-1.11.5.tar.gz (1.8Mb): 1.8Mb downloaded
  Running setup.py egg_info for package virtualenv

    warning: no previously-included files matching '*' found under directory 'do
cs/_templates'
    warning: no previously-included files matching '*' found under directory 'do
cs/_build'
Installing collected packages: virtualenv
  Found existing installation: virtualenv 1.11.4
    Uninstalling virtualenv:
      Successfully uninstalled virtualenv
  Running setup.py install for virtualenv

    warning: no previously-included files matching '*' found under directory 'do
cs/_templates'
    warning: no previously-included files matching '*' found under directory 'do
cs/_build'
    Installing virtualenv script to /usr/local/bin
    Installing virtualenv-2.7 script to /usr/local/bin
Successfully installed virtualenv
Cleaning up...
```

Figura 5.1: Salida de la linea de comandos tras la instalación inicial.

```
xt (line 11))
Requirement already satisfied (use --upgrade to upgrade): pyxdg in /usr/lib/pyth
on2.7/dist-packages (from configglue==1.0->-r stable-req.txt (line 20))
Requirement already satisfied (use --upgrade to upgrade): pyopenssl in /usr/lib/
python2.7/dist-packages (from cyclone==1.1->-r stable-req.txt (line 22))
Requirement already satisfied (use --upgrade to upgrade): ipython>=0.10 in /usr/
local/lib/python2.7/dist-packages (from ipdb==0.8->-r stable-req.txt (line 35))
Downloading/unpacking requests==2.1.0 (from twython==3.1.2->-r stable-req.txt (l
ine 47))
  Downloading requests-2.1.0.tar.gz (420Kb): 420Kb downloaded
  Running setup.py egg_info for package requests

Requirement already satisfied (use --upgrade to upgrade): requests-oauthlib==0.4
.0 in /usr/local/lib/python2.7/dist-packages (from twython==3.1.2->-r stable-req
.txt (line 47))
Installing collected packages: requests
  Found existing installation: requests 2.2.1
    Uninstalling requests:
      Successfully uninstalled requests
  Running setup.py install for requests

Successfully installed requests
Cleaning up...
```

Figura 5.2: Salida de la linea de comandos tras la instalación de dependencias.

```
2014-05-11 18:50:08+0200 Log opened.
2014-05-11 18:50:08+0200 Application starting on 8888
2014-05-11 18:50:08+0200 Starting factory <cyclone.web.Application instance at 0
x2341908>
```

Figura 5.3: Mensaje por linea de comandos generado al lanzar el servidor.

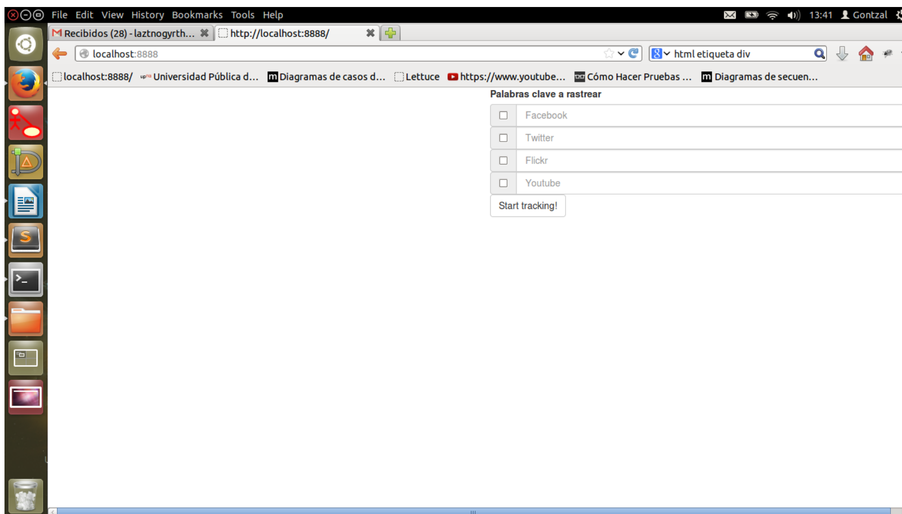

Figura 5.4: Formulario generado por el servidor al acceder vía navegador.

0. Instalar dependencias como se ha indicado en la guía de instalación.

1. Navegar mediante el comando cd de la consola a la ubicación actual de la aplicación.

2. Iniciar el servidor mediante el comando `python server.py`. Se iniciará el servidor y mostrará unos mensajes de log para verificar que se ha iniciado correctamente y decirnos en qué puerto está corriendo el servidor, por defecto, en el 8888. En la Fig. 5.3 se muestra este resultado.

3. Se verifica el éxito de los tres pasos anteriores accediendo a la url asignada (en este caso `http://localhost:8888/`) para verificar que el servidor devuelve el formulario que se incluye en la Fig. 5.4.

5.3. Guía de usuario del cliente/periodista

Una vez se ha lanzado el servidor (siguiendo los pasos descritos en la sección anterior), se puede probar el servidor. El proceso consta de varios pasos:

1. Acceder a la URL proporcionada para la aplicación (ver Fig. 5.5).

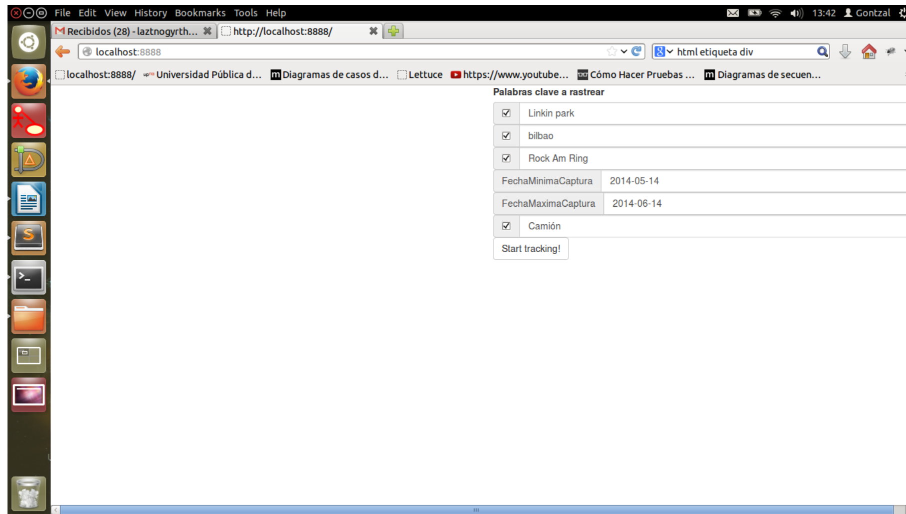

Figura 5.5: Formulario de acceso generado por el servidor.

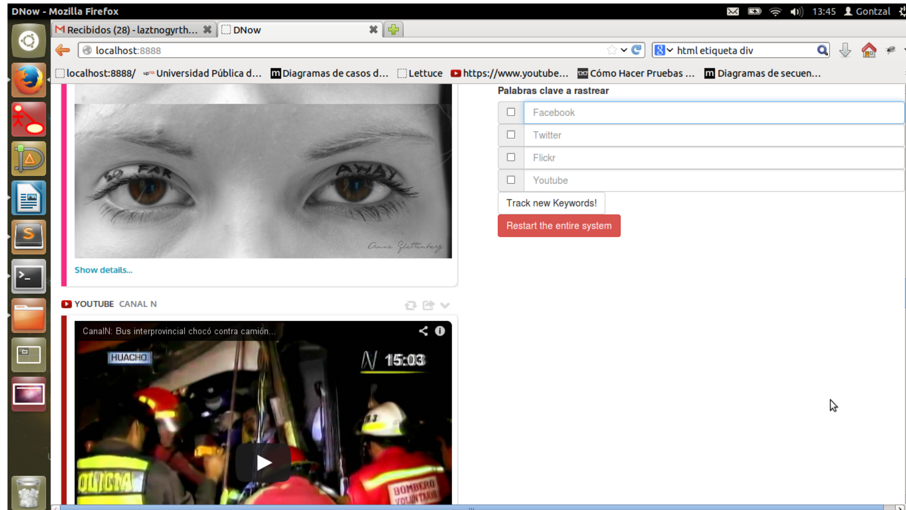

Figura 5.6: Ejemplo de pantallazo generado por el servidor tras una consulta.

2. Introducir en el formulario los términos de búsqueda para las distintas redes sociales y en que redes se ha de buscar.

3. Enviar el formulario y esperar las respuestas de las redes seleccionadas (en el ejemplo visualizado en la Fig. 5.6, estas son Twitter y Youtube).

4. Si se requiere, se pueden cambiar los términos de búsqueda y empezar una nueva si se rellena el formulario y se envía.

Capítulo 6

Problemas encontrados

6.1. Tecnología de parte del servidor y tiempo-real en el cliente

El proyecto, por definición, necesitaba adquirir el HTML de las tarjetas de Embedly en tiempo real. Es decir, necesitaba poder establecer una conexión mediante *sockets* o alguna otra técnica con el servidor. Al principio, barajé utilizar `socket.io` para realizar este cometido, pues es una buena librería de Javascript que permite acciones en tiempo real en aplicaciones web.

Existían otras alternativas como por ejemplo, Tornado. Tornado es un *framework* web para Python que proporciona un conjunto de librerías de red asíncronas que permiten escalar fácilmente para poder aguantar un número considerable de conexiones abiertas simultáneamente. De esta manera es ideal para combinarse junto a WebSocket, que ha sido el protocolo que he elegido para realizar el manejo del tiempo real y no es más que un protocolo que provee canales de comunicación *full-duplex* (en ambos sentidos) sobre una única conexión de TCP. Además, Websocket permite flujos de mensajes (*streams*) sobre TCP. Antes de Websocket, para alcanzar la comunicación full-duplex sobre el puerto 80, era necesario utilizar canales Comet. Sin embargo, la implementación de Comet no es trivial, y no es eficaz para sistemas que envían mensajes pequeños. El protocolo de Websocket tiene como objetivo resolver estos problemas sin comprometer la seguridad de la web.

Twisted es un *framework* de red para la programación dirigida por eventos y escrito en Python. Proporciona soporte para varias arquitecturas (TCP, UDP, SSL/TLS, IP Multicast...) y un gran número de protocolos (como HTTP, XMPP, NNTP, IMAP, SSH, IRC, FTP).

Al final, opté por Cyclone. Cyclone es un *framework* de servidor web para Python que implementa la API de Tornado como un protocolo de Twisted. La idea es tender un puente entre la API de Tornado y la programación dirigida por eventos de Twisted, soportando un gran número de protocolos. Cyclone nos permite escribir, fácilmente, un servidor consistente en clases que definen el comportamiento del sistema respecto a las peticiones entrantes como `GET` y `POST` (ejemplo en la Fig. 6.1). De esta manera, también podemos soportar la abstracción de tiempo real mediante un poco de código Javascript

```
@cyclone.web.authenticated  # triggers authentication
@defer.inlineCallbacks      # allows for inline callbacks
def get(self):

    result = yield self.do_download()
    self.write(result)
```

Figura 6.1: Servidor de mensajes **POST/GET** construído sobre Cyclone.

funcionando con Websocket.

6.2. Problemas en los escuchadores

El sistema dispone de cuatro escuchadores para activar la búsqueda de información. Los datos que se reciben de cada uno de ellos, obviamente, son de una estructura diferente y la manera de tratarlos ha sido distinta. En esta sección se detallan las decisiones escogidas para cada uno.

6.2.1. Escuchador de Twitter

El escuchador de Twitter fue el primero que programé y, por extensión, el que utilizé de modelo para construir el resto de los escuchadores.

El primer aspecto a tener en cuenta fue el flujo de mensajes que puede tener Twitter sobre un tema «X». Es obvio que las personas que están afiliadas a Twitter tienen una tendencia a publicar muchos más *tweets* que publicaciones en Facebook, fotos en Flickr o cualquier otra actividad habitual en otra red social. Se puede deducir fácilmente que el flujo de tweets escritos para un tema «X» es, sin duda, mayor al flujo que podemos obtener de otras redes sociales. Por ejemplo con un tema como una banda de música famosa (por ejemplo Linkin Park) se pueden publicar aproximadamente de 4 a 7 tweets cada 10 segundos, sin ni siquiera ser *trending topic* en ese momento. Un *trending topic* puede, por tanto, generar una cantidad de tweets ingente por segundo.

La primera opción que estudié fue la de utilizar la típica Search API que proporciona Twitter. Es parte de la v1.1 de la REST API de Twitter y permite realizar consultas acerca de los *tweets* más populares funcionando de manera muy similar al de su buscador por cliente web. El primer problema relacionado con la Search API de Twitter consiste en que la Search API esta enfocada a la relevancia de los *tweets* y no a la búsqueda completa de los mismos. Esto quiere decir que ciertos *tweets* y/o usuarios son omitidos al realizar una búsqueda. El segundo problema es obvio respecto a lo que he explicado en el párrafo anterior y con respecto a lo que se pidió que debía hacer el sistema. Se necesita que los *tweets* sean en tiempo real, y utilizando la Search API de Twitter esto significa que debemos hacer llamadas cada poco tiempo, realizar una búsqueda del ultimo *tweet* que conseguimos en la búsqueda anterior para no enviar a nuestro cliente *tweets* repetidos y enviarle aquellos que no se hubieran visto.

Dados estos dos significativos problemas de la Search API, opté por utilizar la Streaming API de Twitter. La Streaming API básica que ofrece Twitter a los desarrolladores es un acceso limitado al *stream* global de datos de Twitter. De entre los diferentes *endpoints* que ofrece el Streaming API opté por el *stream* público porque es el más adecuado para rastrear distintos usuarios, temas y para la minería de datos en general. Una vez que la aplicación establece una conexión con el *endpoint*, se le va entregando un *feed* de *tweets*. Por otra parte, podría haber elegido el endpoint de *site streams* que contiene todo el tráfico de *tweets* de Twitter (a diferencia del *stream* público que solo recibe parte del tráfico de *tweets* cuando el tema a buscar es *trending topic*) pero, es necesario pedir permiso a Twitter y además, no proporciona ningún filtro como *track* o *keywords* (el *stream* público sí) de manera que complicaba demasiado el desarrollo del escuchador de Twitter.

Así pues, al final, decidí implementar el escuchador de Twitter mediante una libreria llamada Tweepy que permite un acceso sencillo y limpio al *stream* público de Twitter. De hecho, el código para iniciar el escuchador se resume en:

```
twitterStream = Stream(auth, self)

twitterStream.filter(track=self.track,async=self.async)
```

6.2.2. Escuchador de Flickr

Habiendo pasado todo el periodo de pruebas y problemas de Embedly con la Stream API de Twitter, implementar el resto de escuchadores fue una tarea un poco más sencilla.

A diferencia de Twitter, Flickr pone a disposición de los usuarios una única API que funciona de manera similar a como lo hace la Search API de Twitter. Es lógico pensar que el flujo de fotos que se van a subir con un tag «X» va a ser mucho menor que el flujo de *tweets*, como ya he comentado antes. De esta manera no tenemos la necesidad de un Stream API de Flickr ya que podemos hacer llamadas cada minuto (para no saltarnos las limitaciones que nos impone la API de Flickr) y realizar una búsqueda de la ultima foto encontrada en la ultima búsqueda para comprobar si se han subido fotos nuevas desde la ultima vez que realizamos una petición a la API. Ahí es donde encontré el primer problema con la API de Flickr. Si accedemos a su explorador de API, podemos realizar pruebas de como funciona la API y de los resultados que devuelve. Realizando una prueba con el tag «bilbao» y ordenado de manera descendente por la fecha en la que se publico la foto, obtenemos resultados distintos en dos llamadas consecutivas como se puede observar en la Fig. 6.2.

El resultado de la Fig. 6.2, obviamente, está desordenado. Flickr, al devolver estos ficheros en formato JSON, no ordena totalmente por el tiempo exacto si no que lo ordena por hora y usuario, saltándose los minutos y los segundos. Es decir, si un usuario sube a la vez varias imágenes, cuando realicemos varias llamadas para obtener esas imágenes, las respuestas estarán, probablemente, desordenadas respecto a otras llamadas. Esto ha complicado la lógica del escuchador de Flickr.

Consideré una manera bastante sencilla para solucionar el problema anterior y no darle carga al servidor con algoritmos de ordenación en el mismo. La idea consiste en guardar el *id* de la ultima imagen vista y el *id* del ultimo usuario visto. De esta manera,

```
      { "id": "14186184335", "owner": "31928151@N02", "secret": "ca80499b4c", "server": "7389",
"farm": 8, "title": "Audience. BasqueFest", "ispublic": 1, "isfriend": 0, "isfamily": 0,
"dateupload": "1400089582" },
      { "id": "13999532588", "owner": "31928151@N02", "secret": "2e62e95eb3", "server": "5587",
"farm": 6, "title": "Audience. BasqueFest", "ispublic": 1, "isfriend": 0, "isfamily": 0,
"dateupload": "1400089580" },
      { "id": "14182883751", "owner": "31928151@N02", "secret": "1707f27c44", "server": "7341",
"farm": 8, "title": "Audience. BasqueFest", "ispublic": 1, "isfriend": 0, "isfamily": 0,
"dateupload": "1400089584" },
      { "id": "13999528260", "owner": "31928151@N02", "secret": "c9829df575", "server": "7401",
"farm": 8, "title": "Audience. BasqueFest", "ispublic": 1, "isfriend": 0, "isfamily": 0,
"dateupload": "1400089579" },


      { "id": "13999532588", "owner": "31928151@N02", "secret": "2e62e95eb3", "server": "5587",
"farm": 6, "title": "Audience. BasqueFest", "ispublic": 1, "isfriend": 0, "isfamily": 0,
"dateupload": "1400089580" },
      { "id": "14182883751", "owner": "31928151@N02", "secret": "1707f27c44", "server": "7341",
"farm": 8, "title": "Audience. BasqueFest", "ispublic": 1, "isfriend": 0, "isfamily": 0,
"dateupload": "1400089584" },
      { "id": "14186184335", "owner": "31928151@N02", "secret": "ca80499b4c", "server": "7389",
"farm": 8, "title": "Audience. BasqueFest", "ispublic": 1, "isfriend": 0, "isfamily": 0,
"dateupload": "1400089582" },
      { "id": "13999528260", "owner": "31928151@N02", "secret": "c9829df575", "server": "7401",
"farm": 8, "title": "Audience. BasqueFest", "ispublic": 1, "isfriend": 0, "isfamily": 0,
"dateupload": "1400089579" },
```


Figura 6.2: Ejemplo de dos llamadas consecutivas a la API de Flickr

al pedir imágenes a Flickr, si se repite el *id* de la imagen, sabemos que esa imagen ya a sido enviada al cliente y si se repite el *id* del usuario, suponemos que el conjunto de imágenes que haya subido el usuario ya ha sido enviado al cliente debido a que no es habitual subir varios conjuntos de fotos a la misma hora. Hay que destacar que, aparte de el orden por horas, la API también respeta un orden por usuario así que si un usuario distinto al ultimo usuario visto por el sistema sube un conjunto de imágenes después de una llamada del servidor, el servidor obtendrá esas imágenes en la siguiente llamada a la API.

6.2.3. Escuchador de Youtube

En cuanto al escuchador de Youtube se refiere, la tarea fue bastante más sencilla que en el escuchador de Flickr porque la API de Youtube sí que devuelve los resultados ordenados totalmente por fecha, en caso de que se le exija.

Youtube ofrece cuatro API's básicas: la *API de datos*, la *Live streaming API*, la *Analytics API* y la *API del reproductor*. La Live streaming API está diseñada para crear, actualizar y manejar eventos en vivo en Youtube como por ejemplo los streaming de videos que son tan populares hoy en día. La API de YouTube Analytics permite que una aplicación recupere estadísticas de reproducción, métricas de popularidad e información demográfica de los videos y canales de YouTube. La API del reproductor permite controlar cómo se ven los vídeos de YouTube en una web externa. Permite insertar dos tipos de reproductores: el normal y el reproductor "sin bordes", que no es más que un cuadro de vídeo que no tiene ningún control. El reproductor sin bordes está diseñado para programadores

web con experiencia que quieren diseñar su propio reproductor de vídeo.

La API de datos, permite a una aplicación realizar muchas de las operaciones disponibles en la propia página web de YouTube. Es posible realizar búsquedas de vídeos, recuperar *feeds* estándar y ver contenido relacionado. Una aplicación también se puede autenticar como un usuario para subir vídeos y modificar las listas de reproducción de un usuario, entre otras muchas cosas. La API de datos (actualmente en la v3) está diseñada para desarrolladores acostumbrados a programar en lenguajes de servidor y resulta muy útil para aquellos sitios que necesiten una mayor integración con Youtube, como es nuestro caso, al permitir embeber los videos en el cliente.

Por todo lo anterior, es evidente que la mejor opción era la API de datos, pues permite obtener datos acerca de los usuarios, de los videos, de las listas de reproducción y en definitiva de todo lo que necesitaba saber sobre Youtube. Sin embargo, persiste el problema que apareció en Flickr. No dispone de una API que proporcione un *stream* de datos, así que debemos realizar llamadas a la API de datos cada cierto tiempo y realizar un búsqueda del último video que hubiéramos visto en la búsqueda anterior. Esto resulta trivial debido a que devuelve los datos ordenados de manera descendente por publicación mediante un parámetro en la llamada: `query.orderby='published'`.

En definitiva, lo único que hace el escuchador es pedir datos a Youtube sobre la(s) palabra(s) clave que se le han exigido rastrear. Tras ello, debe recoger la lista que le envía la API de datos, enviar los elementos de la lista (eventualmente al cliente manteniéndose en pausa durante unos segundos tras cada iteración) y al terminar la lista, volver a solicitar videos a Youtube para comprobar si se han subido nuevos videos etiquetados con el tema en cuestión.

6.2.4. Escuchador de Facebook

La principal API que ofrece Facebook (y la que he usado) es la Graph API. Aparte de esta API , existen otras cuatro APIs para desarrolladores:

- *Public feed API:* La API de *feed* «público» de Facebook ofrece un *stream* de datos de las actualizaciones de usuario y páginas públicas al instante en el que se publican en Facebook. Esta API sería la ideal a utilizar pero, sin embargo, el acceso a esta API está restringido a un pequeño conjunto de creadores o editores de contenido multimedia y su uso requiere un acuerdo con Facebook.

- *Keyword Insights API:* La Keyword Insights API expone una *cpa* de análisis en la parte superior de todos los posts realizados en Facebook que permite consultar distintas ideas anónimas sobre gente que habla de un determinado tema.

- *FQL:* El Facebook Query Language, o FQL, permite utilizar una interfaz de estilo SQL para consultar los datos expuestos por la Graph API de Facebook. Provee algunas características avanzadas que no están disponibles en la *Graph API*, como por ejemplo utilizar el resultado de una consulta en otra consulta.

- *Chat API:* Sirve para integrar el sistema de chat de Facebook en nuestros productos de mensajería. Se conecta al chat de Facebook a través de del servicio Jabber XMPP.

Por otra parte, la actual *Graph API* de Facebook se encuentra en la v.2 y al pasar de la v.1 a la v.2 la búsqueda de posts quedó desaprobada. Si se llama a la búsqueda de posts se nos devuelve un mensaje de tipo `(#11) Post search has been deprecated` La única manera que he encontrado para solventar esto es hacer llamadas a la v.1 de la API. Para ello debemos hacer peticiones en la dirección `https://graph.facebook.com/v1.0/search` con una *query string* tal que

```
?q=search_query&type=post&access_token=access_token
```

Nótese que es necesario un `access_token` generado por una aplicación registrada antes del cambio a la API v.2. En mi caso, lo he obtenido de una aplicación que registré cuando empece a hacer pruebas de minería social en Facebook.

Una vez tenemos el `access_token` el procedimiento es bastante similar al de Flickr y al de Youtube. Pedimos una serie de *posts* que contengan el *tag* «X» y nos guardamos la *id* del primero (nos lo devuelve los resultados ordenados por fecha) que no hayamos almacenado en la cache con anterioridad. Después recorremos la respuesta (a la que se le puede poner un limite para no sobrecargar al servidor) hasta encontrar el ultimo *id* que obtuvimos en la anterior petición o hasta recorrer toda la respuesta. De esta manera, saldremos del bucle y tendremos los *id* almacenados en un lista que pasaremos al método `updatePosts` para que se encargue de generar las tarjetas de Embedly y mandarlas a la caché para actualizarla y actualizar a los clientes.

6.3. Problemas en el uso de Embedly

Embedly es un conjunto de herramientas que ayudan a los desarrolladores de aplicaciones de escritorio y web a crear mejores experiencias de usuario. Las mejores bazas de Embedly son su sencillez de uso y la estética estandarizada que devuelve para un gran número de proveedores (entre los que se encuentran las redes sociales que rastreo). Desde su propia página de inicio podemos pedir que nos muestre como formatea las peticiones (ver Fig. 6.3).

La manera mas sencilla para conseguir este tipo de tarjetas HTML a partir de los *tweets*, fotos, post, videos... es introducir la URL de cada uno en un ancla de HTML de la siguiente manera:

```
<a class=”embedly-card” href=”URLDELTWEET_FOTO_VIDEO”>NombreX</a>.
```

Para ello, sólo debo conseguir la URL de cada pieza de información. Esto se hace de manera distinta para cada uno de los medios sociales considerados:

- *Twitter:* Devuelve los datos en un `string` que se puede formatear a JSON. Una vez hecho esto, obtengo la *id* del usuario del JSON así como el nombre del usuario y construyo la URL del *tweet* de la siguiente manera:

```
tweet_URL =”https://twitter.com/”+tweet_user+”/status/”+tweet_id.
```

- *Flickr:* Utilizando la librería de Flipy se obtienen los datos en formato de objeto de Python. Obtengo el usuario y el *id* de la foto para construir la URL:

```
flickrURL=”http://www.flickr.com/photos/”+str(photo.owner)+”/”+str(photo.id).
```

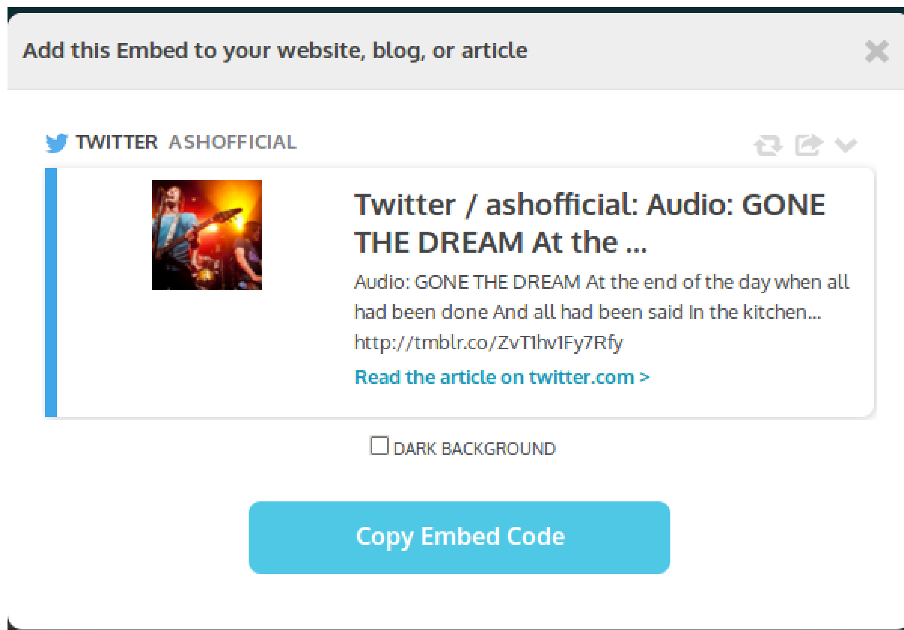

Figura 6.3: Ejemplo de tarjeta HTML producida con Embedly.

- *Youtube:* Youtube devuelve el *id* del video en formato de una URL de este tipo:

  `http://gdata.youtube.com/feeds/videos/id,`

  Así que tuve que utilizar una expresión regular para obtener el *id* del video:
  `r = re.split('http://gdata.youtube.com/feeds/videos/',idStr),`

  seguido de

  `videoId = r[1].`

  Después de esto, la URL del video (`ytURL`) se construye fácilmente:

  `ytURL="http://www.youtube.com/v/"+videoId.`

El tiempo de renderizado de las tarjetas HTML a resultado un problema. Embedly no tiene problemas para renderizar rápidamente tweets o publicaciones de Facebook pero, cuando se trata de renderizar imágenes, el tiempo puede ascender a unos 7 segundos (tiene que realizar escalados de imagen). Para solventar este problema respecto al periodista, las tarjetas HTML esperan sin mostrarse durante 8 segundos. De esta manera les da tiempo a renderizarse y no producen ninguna molestia visual al periodista.

6.4. Rastrear términos que incluyen ñ o acentos

El número de acentos y palabras con el caracter «ñ» que contiene la lengua castellana presentan un problema a la hora de que un periodista elija términos que rastrear en las redes sociales. Si no le indicamos lo contrario, Python, por defecto, contempla únicamente la codificación en ASCII de 7 bits en los que no se incluye ni la «ñ», ni los

acentos, ni ningún otro carácter que no esté en el teclado americano. Para solventar este problema, la solución general por la que he optado ha sido codificar el mensaje obtenido desde los formularios a UTF-8 para después poder enviar lo obtenido a las APIs correspondientes (exceptuando la de Twitter que necesita que se haga una decodificación explícita de UTF-8).

6.5. Archivo de configuración

Añadir o quitar escuchadores resultaba tedioso a la hora del desarrollo. Por eso, incluí un archivo de configuración muy básico en XML para poder modificar de manera rápida qué escuchadores hay disponibles y con qué opciones pueden ser lanzados. De esta manera, se facilita el añadir o quitar escuchadores de una manera elegante y sencilla.

El *parseador* de XML lee el archivo de configuración cada vez que se le llama, traduce la información contenida en él y devuelve en un diccionario de Python todas las redes sociales disponibles con sus opciones, si las hubiera (las claves son el nombre de la red social y los atributos de cada clave las opciones que tiene esa red social si es que tienen alguna). Un ejemplo de archivo de configuración valido es: `<Config> <SocialNetworks version="0.1» <Twitter> </Twitter> </SocialNetworks> </Config>` El utilizado para este proyecto se encuentra en la Fig. 4.12.

Capítulo 7

Pruebas del sistema

7.1. Pruebas de integración

Estas pruebas se realizaron con Lettuce 0.19. Lettuce es una herramienta para Behavior Driven Development (BDD), y que se puede utilizar, entre otras cosas, para realizar pruebas de integración. Este es precisamente el uso que le he dado en el presente proyecto. El uso de Lettuce se establece en una secuencia de cuatro pasos básicos: (a) describir el comportamiento, (b) definir los pasos en Python, (c) verlo fallar y (d) escribir el código que lo hace funcionar:

- Describir el comportamiento: Aunque soporta varios idiomas, es recomendable describir el comportamiento en Inglés. Para ello, describimos el comportamiento deseado en un archivo con extensión `.feature`, como el que se incluye en la Fig. 7.1. De esta manera describimos que queremos introducir el HTML “<p>Hi!</p>” en la cache(actualizandola) y que podemos observar el cambio.

```
Feature: Integration tests
      In order to see if cache is well

      Scenario: Update cache
            I have the html "<p>Hi!</p>"
            I update the cache
            I can see somewhere the html "<p>Hi!</p>"
```

Figura 7.1: Ejemplo de fichero `feature` para la configuración de pruebas.

- Definir los pasos en Python: Para que Lettuce entienda el escenario que le hemos propuesto, tenemos que crear un archivo `steps.py` que describa los pasos (ver Fig. 7.2). Se observa como lo único que hacen los pasos de este ejemplo es crear una instancia de la caché, actualizarla y mirar si de verdad se ha actualizado.
- Verlo fallar: Es obvio que este caso va a fallar porque "Html de prueba de fallo"no es un HTML que este contenido en la cache.

```
from lettuce import *
@step('I have the html "(.*)"')
def have_the_html(step, html):
      world.html = str(html)
      world.cache = embedCache()
@step('I update the cache')
def update_the_cache(step):
      html={}
      html['html']=world.html
      world.cache.update_cache(html)

@step('I can see somewhere the html "(.*)"')
def see_somewher_html(step, html):
      assert_equals(str(html),"Html de prueba de fallo")
```

Figura 7.2: Fichero de descripción de procedimiento para Lettuce.

- Hacerlo funcionar: Para este caso tan sencillo bastaría con cambiar el ultimo paso a como se muestra en la Fig. 7.3, para que devuelva el resultado correcto.

```
@step('I can see somewhere the html "(.*)"')

      def see_somewher_html(step, html):
      assert_equals(str(html),world.cache.cache[0])
```

Figura 7.3: Ejemplo de método correcto para la comprobación de la caché.

De esta manera, obtendríamos el resultado esperado. Por ejemplo, he realizado la prueba que se incluye en la Fig. 7.4 para verificar que todos los escuchadores ven los cambios realizados en la cache. En esta prueba, obtengo el nombre de los escuchadores definidos (Youtube, Facebook, Twitter) y creo la caché que los iniciará, después inicio y paro inmediatamente los escuchadores (para que no devuelvan mensajes de rastreo). Por último, cambio el contenido de la caché manualmente y compruebo uno por uno que todos ven los cambios en la caché gracias a que cada uno de ellos tiene una referencia a la caché.

7.2. Pruebas de carga con ApacheBench

ApacheBench es una herramienta de uso gratuito que proporciona la fundación Apache. En Mac OS X viene preinstalada por defecto, pero para sistemas Linux basados en Debian nos basta con hacer un:

```
sudo apt-get install apache2-utils
```

ApacheBench permite realizar numerosas pruebas de distinto tipo. Permite especificar el número de peticiones que deseamos enviar, el número de conexiones concurrentes y además, permite generar una gráfica de Gnuplot para apreciar mejor los resultados. He realizado varias pruebas en contra del servidor Cyclone implementado.

```
Scenario: Everyone sees the changes
  I have the listeners:
      | name      |
      | Youtube   |
      | Facebook  |
      | Twitter   |
  I change cache with "Data changed"
  And everyones see the change

      @step('I have the listeners:')
      def have_the_html(step):
              ne=[]
              i=0
              for dicti in step.hashes:
                      ne.append(step.hashes[i]['name'])
                      i=i+1
              world.cache = embedCache()
              world.ne=ne
      @step('I change cache with "(.*)"')
      def change_cache(step,new):
              world.new.append(new)
              world.cache.cache=world.new
      @step('And everyones see the change')
      def change_cache(step):
              world.cache.runListeners("kw",world.ne)
              world.cache.stopListeners()
              for lis in world.cache.listenersInstances:
              print world.cache.listenersInstances[lis].cache.cache[0]
              print "hola"
         assert_equals(world.cache.listenersInstances[lis].cache.cache[0],
                                world.new[0])
```

Figura 7.4: Prueba realizada en Lettuce.

```
Concurrency Level:      100
Time taken for tests:   2.889 seconds
Complete requests:      2500
Failed requests:        0
Write errors:           0
Total transferred:      15467500 bytes
HTML transferred:       14995000 bytes
Requests per second:    865.23 [#/sec] (mean)
Time per request:       115.576 [ms] (mean)
Time per request:       1.156 [ms] (mean, across all concurrent requests)
Transfer rate:          5227.70 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0   19 135.6      0    1000
Processing:    20   46  80.2     40    2835
Waiting:       20   46  80.2     40    2835
Total:         20   65 158.7     41    2836
```

Figura 7.5: Resultados de las pruebas de carga con 2500 peticiones.

Hasta las 2500 peticiones el servidor responde sin problemas a todas ellas y a partir de 2500 empieza a fallar. Los resultados con 2500, 5000 y 10000 peticiones se encuentran en las Fig. 7.5-7.7. Podemos comprobar como aunque el número de peticiones suba, el número de peticiones fallidas no depende unicamente del número de peticiones. Con 10000 peticiones, el orden de fallos es desde 0 a 8000 situándose en 4000 de media, aproximadamente. Teniendo en cuenta que se suponen no más de 20 conexiones activas, podemos concluir que no será un gran problema el las condiciones normales de uso. La gran tasa de fallos a partir de las 2500 peticiones es causada, probablemente, por la incapacidad de Embedly de responder a todas las peticiones al mismo tiempo. Superadas de las 2500 peticiones, se realizan un mínimo de 50000 peticiones a Embedly, incrementando la probabilidad de fallo en gran medida.

Por otra parte, se ha de tener cuidado a la hora de buscar términos. Si se buscan temas muy destacados en Twitter, se produce una incapacidad de ver los resultados. Es decir, si la API de Twitter nos envia tweets relacionados a un ritmo mayor al que se pueden renderizar (recordemos que les cuestan renderizar de 2 a 6 segundos y mostrarse 7segundos) y visualizar, el periodista verá la columna de las tarjetas HTML vacía.

```
Concurrency Level:      100
Time taken for tests:   3.347 seconds
Complete requests:      5000
Failed requests:        3282
  (Connect: 0, Receive: 0, Length: 3282, Exceptions: 0)
Write errors:           0
Total transferred:      30228576 bytes
HTML transferred:       29283576 bytes
Requests per second:    1494.02 [#/sec] (mean)
Time per request:       66.934 [ms] (mean)
Time per request:       0.669 [ms] (mean, across all concurrent requests)
Transfer rate:          8820.70 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0   10  99.4      0    1000
Processing:    19   56  21.9     60     545
Waiting:       19   56  21.9     60     545
Total:         19   66 103.8     60    1277
```

Figura 7.6: Resultados de las pruebas de carga con 5000 peticiones.

```
Concurrency Level:      100
Time taken for tests:   6.667 seconds
Complete requests:      10000
Failed requests:        2119
  (Connect: 0, Receive: 0, Length: 2119, Exceptions: 0)
Write errors:           0
Total transferred:      60576012 bytes
HTML transferred:       58686012 bytes
Requests per second:    1499.85 [#/sec] (mean)
Time per request:       66.673 [ms] (mean)
Time per request:       0.667 [ms] (mean, across all concurrent requests)
Transfer rate:          8872.55 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    5  69.1      0    1002
Processing:    19   62  17.4     65     292
Waiting:       19   61  17.4     65     292
Total:         19   66  78.3     65    1289
```

Figura 7.7: Resultados de las pruebas de carga con 10000 peticiones.

Capítulo 8

Resumen y Conclusiones

8.1. Resumen

En resumen, el proyecto que he realizado es un módulo de una aplicación que quiere desarrollar Diario de Navarra. La aplicación debe dar soporte a un periodista para que pueda hacer llegar información acerca de lo que se esta comentando en las redes sociales de un tema en concreto y noticias anteriores relacionadas de manera semántica con ese tema. Mi proyecto en particular se centra en las redes sociales, es decir, rastrea en las mismas una serie de términos que introduce un periodista en un formulario. El periodista puede elegir sobre que redes sociales se ha de realizar la búsqueda de dichos términos. Las opciones disponibles incluyen:

- *Twitter:* Se buscan los términos en tiempo real sobre la Stream API de Twitter. Esto implica que solo se obtienen *tweets* a partir del momento que se empieza a buscar, y no antes. Tan pronto como un usuario de Twitter *twittee* un texto que incluya las palabras que estamos buscando, la API nos enviara los datos de ese *tweet*.
- *Flickr:* El formulario nos da la opción de introducir una fecha de captura mínima y una fecha de captura máxima. En el caso de no introducirlas, se buscan las imágenes subidas entre ayer y hoy. Después de un tiempo que define un operario de Diario de Navarra, vuelve a hacer una petición para evaluar si existen nuevas imágenes que enviar al periodista.
- *Youtube:* De forma similar al buscador de Flickr, realiza una búsqueda estática sobre los videos de Youtube y vuelve a realizar la búsqueda eventualmente para analizar si existen nuevos videos relacionados con la búsqueda.
- *Facebook:* Al igual que los escuchadores de Flickr y Youtube, el escuchador de Facebook realiza una búsqueda estática realizando cada cierto tiempo una nueva llamada para buscar nuevos *posts* relacionados con el tema.

Los mensajes que recibe la aplicación se envían al periodista y antes de mostrarse, realizan una petición a un sistema externo llamado Embedly para formatear de una

manera estándar todos los *posts*, *tweets*, videos o imágenes. El periodista está conectado en todo momento al servidor mediante una conexión de Websocket permitiendo el paso de mensajes en tiempo real y sin necesidad de recargar la página.

8.2. Conclusiones

Tras el desarrollo de esta aplicación he conseguido sacar una serie de conclusiones:

- La implicación por parte del cliente durante los procesos de análisis, diseño y desarrollo es vital, marcando la diferencia entre un proyecto bien definido y que se ajuste a los requisitos y un proyecto a medias, con requisitos ambiguos etcétera.
- Hay que tener en cuenta los sistemas externos a la hora de valorar las condiciones de error de los casos de uso. Muchas veces tendemos a dar por hecho que funcionan siempre al 100 % y no es así.
- Se han de estudiar y analizar todas las opciones disponibles(o gran parte de ellas) para solventar un problema o necesidad.
- Ninguna de las APIs que las redes sociales ponen a disposición de los desarrolladores funciona de la misma manera(obvio sin pensamos que son diferentes empresas) y es necesario estudiarse cada una para poder realizar el mejor uso de ellas.
- La cantidad de información que se puede extraer de *tweets*, imágenes, *posts* o videos es ingente. Seria muy interesante poder realizar algoritmos de búsqueda más avanzados y combinarlos junto a la búsqueda semántica que ha realizado mi compañero.
- Se ha de buscar con cuidado. Un *trending topic* de Twitter puede colapsar fácilmente el servidor o en una forma menos drástica impedir que el periodista vea nada. En todo momento el periodista puede ver como mucho veinte tarjetas HTML. Si el flujo combinado de las redes sociales produce tarjetas a un ritmo mayor del que les cuesta renderizarse, no se vera ninguna tarjeta HTML y se debe tener cuidado y precisión a la hora de introducir los términos a buscar.

Capítulo 9

Bibliografía

[1]: Matthew A. Russell, Mining the Social Web, 2nd Edition

[2]: Maksim Tsvetovat & Alexander Kouznetsov, Social Network Analysis for Startups

[3]: http://danielnill.com/nodejs-tutorial-with-socketio/

[4]: https://twistedmatrix.com/documents/12.3.0/web/howto/using-twistedweb.html#auto0

[5]: http://twistedmatrix.com/documents/current/web/howto/web-in-60/index.html

[6]: http://twistedmatrix.com/documents/10.1.0/web/howto/client.html

[7]: https://dev.twitter.com/docs/api/1/get/statuses/oembed

[8]: http://halotis.com/2009/09/08/download-images-from-flickr-with-python/

[9]: http://codehero.co/como-hacer-pruebas-de-carga-servidores-web/

[10]: http://lettuce.it/tutorial/simple.html

[11]: http://cyclone.io/documentation/

[12]: http://cyclone.io/documentation/websocket.html

[13]: http://www.tornadoweb.org/en/stable/documentation.html

[14]: http://tools.ietf.org/html/rfc6455

Índice de figuras

Introducción 7

1.1. Ejemplo de tarjeta HTML. 11

Requisitos y arquitectura 13

2.1. Ejemplo de clase para la gestión de mensajes con Cyclone. 14
2.2. Diagrama de componentes general del sistema. 15
2.3. Interfáz gráfico propuesto en el prototipo de aplicación. 16

3.1. Diagrama de casos de uso del sistema 17
3.2. Diagrama de actividad para la inicialización del sistema. 19
3.3. Diagrama de actividad para la creación de un formulario. 20
3.4. Diagrama de actividad para el rellenado de un formulario. 21
3.5. Diagrama de actividad para inicializar los escuchadores. 22
3.6. Diagrama de actividad para mostrar la caché. 24
3.7. Diagrama de actividad para la actualización del sistema. 28
3.8. Diagrama de actividad la reinicialización de los escuchadores. 29
3.9. Diagrama de actividad la reinicialización del sistema. 30
3.10. Diagrama de secuencia del cliente. 31

4.1. Diagrama de componentes específico del sistema. 34
4.2. Diagrama de actividad del componente servidor 35
4.3. Diagrama de actividad del componente escuchadores 36
4.4. Diagrama de actividad del componente cliente 37
4.5. Interfaz de inicio. 38
4.6. Interfaz de inicio en el proceso de introducir datos. 38
4.7. Interfaz con escuchadores activos. 39
4.8. Diagrama de clases del componente escuchadores. 40
4.9. Diagrama de clases del componente servidor. 41
4.10. Servidor de *echo* implementado sobre WebSocket. 41
4.11. Código JavaScript de cliente WebSocket 41
4.12. Contenido del archivo de configuración. 42
4.13. Diagrama de despliegue del sistema. 43

Uso del sistema **45**
5.1. Salida de la linea de comandos tras la instalación inicial. 46
5.2. Salida de la linea de comandos tras la instalación de dependencias. 46
5.3. Mensaje por linea de comandos generado al lanzar el servidor. 47
5.4. Formulario generado por el servidor al acceder vía navegador. 47
5.5. Formulario de acceso generado por el servidor. 48
5.6. Ejemplo de pantallazo generado por el servidor tras una consulta. 48

Problemas encontrados **51**
6.1. Servidor de mensajes **POST/GET** construído sobre Cyclone. 52
6.2. Ejemplo de dos llamadas consecutivas a la API de Flickr 54
6.3. Ejemplo de tarjeta HTML producida con Embedly. 57

Pruebas del sistema **59**
7.1. Ejemplo de fichero `feature` para la configuración de pruebas. 59
7.2. Fichero de descripción de procedimiento para Lettuce. 60
7.3. Ejemplo de método correcto para la comprobación de la caché. 60
7.4. Prueba realizada en Lettuce. 61
7.5. Resultados de las pruebas de carga con 2500 peticiones. 62
7.6. Resultados de las pruebas de carga con 5000 peticiones. 63
7.7. Resultados de las pruebas de carga con 10000 peticiones. 63

Resumen y Conclusiones **65**