Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Diseño de aplicaciones multi-paradigma en Erlang

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Óscar Muñoz Garrigós
Tutor: Germán Francisco Vidal Oriola

2015-2016

Resumen

En esta memoria se describe el desarrollo de una aplicación multi-paradigma, en la que el componente principal de coordinación está implementado en el lenguaje de programación Erlang. Se ha escogido Erlang por su gran capacidad de concurrencia, por su habilidad para diseñar sistemas tolerantes a fallos y por el hecho de poder mantener y actualizar los componentes del sistema sin realizar paradas en la ejecución. En este proyecto, exploramos también la interacción entre Erlang y Java, uno de los lenguajes más utilizados en sistemas operativos como Android, a la hora de desarrollar una aplicación de mensajería instantánea.

Palabras clave: Concurrencia, Erlang, Programación, Ingeniería del software, Programación funcional, Android**.**

Abstract

In this report, we describe the development of a multi-paradigm application, in which the main coordination component is implemented in the programming language Erlang. Erlang was chosen for its ability for scalable concurrency, fault-tolerant designs, and hot code uploading. In this project, we explore the interaction between Erlang and Java, one of the most used languages on operating systems like Android, to develop an instant messaging application.

Keywords: Concurrency, Erlang, Programming, Software engineering, Functional programming, Android.

Tabla de contenidos

1 INTRODUCCIÓN

Documento realizado para la memoria del Trabajo de Final de Grado de Ingeniería Informática cursado en la Escuela Técnica Superior de Ingeniería Informática de la Universidad Politécnica de Valencia.

1.1 CONTEXTO

Erlang es un lenguaje de programación que puede dar solución a muchos de los problemas que se enfrentan las grandes compañías estos días.

Esta aplicación pretende ser un punto de referencia para aquellos que quieran aprovechar el uso de Erlang en sus aplicaciones, dando a conocer las ventajas que tiene y las dificultades a la hora de comunicarse con otros lenguajes, poniendo como ejemplo una aplicación de mensajería instantánea.

1.2 OBJETIVOS DEL PROYECTO

La finalidad del proyecto es la de crear una aplicación de mensajería instantánea donde poder tener conversaciones con nuestros conocidos y poder gestionar los contactos y las conversaciones que tenemos.

La aplicación deberá cumplir los siguientes requisitos:

- Permitir crear cuentas nuevas a los usuarios que quieran utilizar la aplicación.
- Permitir cargar la cuenta a aquellas personas que ya tengan una cuenta en la aplicación.
- Poder añadir y visualizar contactos en nuestra lista de contactos.
- Poder crear y visualizar chats con nuestros contactos.
- Poder enviar mensajes a nuestros contactos a través de los chats.

1.3 ESTRUCTURA DEL DOCUMENTO

Este documento presenta una estructura en etapas, común en el desarrollo de un proyecto software. Estas etapas son las siguientes:

Especificación de requisitos: En esta fase se reúnen todos los requisitos funcionales que debe tener el producto final.

- Análisis: El análisis describe la estructura y funcionalidad del producto mediante diagramas que permitan comprender el sistema. Se incluyen diagramas de clases UML y casos de uso, que describen los principales comportamientos del portal.
- Diseño e implementación: En esta fase se describen los diferentes niveles que componen la arquitectura de la aplicación, así como las diferentes tecnologías utilizadas. Se describen detalladamente la implementación y el código asociado.
- Conclusiones: En este apartado se señalan las conclusiones finales obtenidas y los objetivos cumplidos.
- Bibliografía: La bibliografía consultada para el desarrollo del proyecto y/o la memoria.
- Anexos: En los anexos se incluye cualquier documento que por su extensión se ha decidido incluir al final de la memoria.

1.4 El lenguaje Erlang

Erlang fue diseñado para escribir programas concurrentes que se ejecutan indefinidamente. Los procesos son ligeros, no tienen memoria compartida, se comunican con el paso de mensajes asíncronos y pertenecen al lenguaje (y no al sistema operativo). Los mecanismos incluidos en el lenguaje simplifican la construcción de *software* para implementaciones de sistemas ininterrumpidos.

El desarrollo inicial tuvo lugar en 1986, en el Laboratorio de Informática de Ericsson. Erlang fue diseñado con un objetivo específico en mente: aportar una mejor forma de programar aplicaciones de telefonía.

Las aplicaciones de telefonía son por naturaleza altamente concurrentes. Las transacciones que manejan son intrínsecamente distribuidas y se espera que el *software* sea altamente tolerante a fallos. La mayoría de procesos están esperando a un evento causado por la recepción de un mensaje o el lanzamiento de un temporizador. Cuando un evento ocurre, el proceso hace una pequeña cantidad de computación, posiblemente envía mensajes a otros procesos y entonces espera el siguiente evento. La cantidad de computación involucrada es muy pequeña.

Tres propiedades de un lenguaje de programación son esenciales en la eficiencia de las operaciones de un lenguaje concurrente y la formación de un sistema altamente concurrente:

- El tiempo en crear un proceso.
- El tiempo en realizar el cambio de contexto entre dos procesos
- El tiempo de copia de un mensaje entre dos procesos

Requisitos de un sistema de programación para sistemas de telecomunicaciones:

1. Manejar un número muy grande de actividades concurrentes.
2. Las acciones tienen que ser realizadas en un plazo de tiempo.
3. Los sistemas están distribuidos en diferentes computadores.
4. Interacción con *hardware*.
5. Sistemas de *software* muy grandes.
6. Admisión de funcionalidad compleja.
7. Operación continúa durante varios años.
8. Mantenimiento de *software* sin parar el sistema.
9. Fuertes requisitos de calidad y fiabilidad.
10. Tolerancia a fallos de *hardware* y de *software*.

Puntos clave de Erlang

- Actualización en tiempo de ejecución: Erlang permite cambiar el código en tiempo de ejecución, en otras palabras, el cambio del código se puede producir sin parar el sistema.
- Buffers: Cada proceso en Erlang tiene su propio buzón de correo donde todos los mensajes entrantes son guardados. Cuando un mensaje entrante llega al buzón de correo, el proceso que lo compara con los patrones es preparado para la ejecución. En el caso de que la comparación con los patrones sea exitosa, la recepción del mensaje ocurre y el mensaje es removido de la lista de correo, ejecutando sus datos en el proceso receptor. Cuando la comparación del mensaje con los patrones falla el mensaje es puesto en la lista de correos guardados. Cuando cualquier otro mensaje es comparado de forma exitosa, todos los correos

guardados en la lista son procesados otra vez por el proceso de comparación de patrones.

- Manejo de errores: En el caso de Erlang todo el *hardware* es tratado como objetos relativos. La única manera de comunicarse con estos objetos es mediante la transmisión de mensajes. Cuando envías un mensaje a un proceso, no debería de haber ninguna manera de conocer si el proceso era un dispositivo físico o simplemente otro proceso *software*. La razón para esto era la simplificación del modelo de programación, de forma que la comunicación con todos los procesos es de forma uniforme. Desde el punto de vista de la programación, Erlang fue diseñado para manejar de la misma forma los errores *software* y los errores *hardware*. El modelo de manejo de errores está basado en la idea de dos computadores que se observan mutuamente. La detección de errores y la recuperación es ejecutada en el ordenador remoto y no en el ordenador local. Esto es debido a que en el peor caso, el ordenador donde el error ha ocurrido ha fallado completamente y no se pueden hacer más computaciones en él.
- Enlaces: Los procesos en Erlang se enlazan los unos a los otros, de forma que puede haber distintos grupos de procesos enlazados entre sí. Cuando un proceso de este grupo falla se emite una señal a todos los procesos del grupo, los cuales liberan todos los recursos solicitados y envían un mensaje avisando del error.
- Máquina virtual: Erlang tiene una máquina virtual sobre la que ejecuta todas sus operaciones. Ésta es la que se encarga de comunicar el lenguaje con el sistema operativo.
- Colector de basura: El colector de basura empezó a ser crucial desde que Erlang se implementa en un sistema de tiempo real. Se escogió por ejecutar el proceso frecuentemente en pequeñas dosis, de esta forma el tiempo de procesamiento es poco y predecible.
- Comportamientos OTP: Para construir cualquier cuerpo significante de software no necesitas simplemente un lenguaje de programación, sino también un conjunto de librerías y un sistema operativo sobre el que ejecutarlo. Además necesitas una filosofía de programación. El nombre colectivo de Erlang, todas las librerías de Erlang el sistema sobre el que se ejecuta y la descripción de las formas en que tiene de hacer las cosas es el sistema OTP. El sistema OTP contiene:
  - Librerías de código Erlang.
  - Patrones de diseño para construcción de aplicaciones comunes.
  - Documentación.
  - Tutoriales.
- Base de datos distribuida (Mnesia): Mnesia es una base de datos implementada especialmente para Erlang la cual satisface los siguientes requisitos:
  - Rápida obtención de tuplas clave-valor.
  - Búsquedas complicadas que no están basadas en tiempo real, principalmente para operaciones y mantenimiento.
  - Datos distribuidos para aplicaciones distribuidas.
  - Alta tolerancia a fallos.
  - Re-configuración dinámica.
  - Objetos complejos.

Tras varios años de investigación en Ericsson, la compañía decide dejar de mantener el lenguaje. Los miembros del proyecto deciden formar una compañía llamada “Bluetail”, haciendo el lenguaje abierto a la comunidad.

2 ESPECIFICACIÓN DE REQUISITOS

2.1 INTRODUCCIÓN

2.1.1 Propósito

La especificación de requisitos tiene como finalidad la descripción completa del comportamiento del sistema que se va a desarrollar. Incluye un conjunto de casos de uso que describen todas las interacciones que tendrán los usuarios con el software.

2.1.2 Definiciones, acrónimos y abreviaturas

- Usuario: La persona que está utilizando la aplicación.
- Contacto: Un conocido del usuario.
- Chat: Ventana de conversación con un contacto.
- Mensaje: Pedazo de texto que el usuario envía a un contacto mediante un chat.

2.1.3 Visión global

El producto a desarrollar es una aplicación de mensajería instantánea.

Los usuarios podrán enviar mensajes a cualquier otro usuario de la aplicación que hayan añadido como contacto previamente.

Para poder enviar mensajes y agregar contactos será imprescindible haberse dado de alta anteriormente.

2.2 DESCRIPCIÓN GENERAL

2.2.1 Perspectiva del producto

La aplicación que desarrollamos es para usuarios de Android. Únicamente aquellos usuarios que tengan el sistema operativo Android instalado en sus dispositivos podrán acceder a nuestra aplicación.

2.2.2 Características de usuarios

Podemos distinguir entre dos tipos de usuarios que van a acceder a la aplicación:

Usuario no identificado: Cualquier persona no identificada que entra en la aplicación.

- Crear cuenta
- Cargar cuenta

Usuario identificado: Aquella persona que accede a la aplicación con usuario y contraseña.

- Añadir contacto
- Modificar contacto
- Eliminar contacto
- Añadir chat
- Modificar chat
- Eliminar chat
- Enviar mensaje

2.3 Requisitos específicos

2.3.1 Requisitos de interfaces externas

Las interfaces del cliente deberán ser simples. Los colores no deberán de causar estrés y deberán de tener un color y un contraste agradables. Los textos en la aplicación deben de ser claramente legibles.

2.3.2 Requisitos funcionales

Para poder enviar y recibir mensajes los dispositivos de los usuarios deben de estar conectados a Internet.

Los usuarios deben de poder enviar mensajes entre sí. Esta es la función principal y característica de la aplicación. Para la recepción de dichos mensajes no será necesario que los dos usuarios estén en conexión de manera simultánea.

Cada usuario debe tener un identificador único en la aplicación, y cada usuario deberá identificarse en la aplicación mediante un sistema de usuario y contraseña. Esto permite asegurar que el contenido del usuario será dirigido únicamente a la persona identificada como ese usuario.

Cualquier persona puede crear una cuenta en la aplicación, no hay ningún requisito ni ninguna restricción para que cualquier persona cree una cuenta.

Los usuarios podrán volver a acceder a su cuenta si cambian de dispositivo o deciden prescindir de la aplicación durante un periodo de tiempo.

Los usuarios podrán añadir a otros usuarios a la lista de contactos, pudiendo escoger también cuando quitarlos de ella. Dichos contactos podrán ser editados para que utilicen un nombre más amigable al usuario.

Se podrá añadir, modificar y eliminar chats. La creación de un chat implicará la adición de un contacto en él, dicho de otra forma, no podrá ser creado un chat sin miembros. El número de miembros en el chat inicialmente será de dos: el creador y el contacto del creador. Las modificaciones del chat estarán destinadas al título.

2.3.3 Restricciones de diseño

La aplicación no enviará imágenes ni ningún otro tipo de dato que no sea texto plano.

La aplicación permitirá la creación de chats de forma personal, por lo que no podrán haber más de dos usuarios en un chat.

Los mensajes no podrán ser recuperados, una vez los mensajes son leídos únicamente prevalecerá la copia en el dispositivo cliente. Esto significa que, si el usuario borra el mensaje o el usuario cambia el dispositivo, no volverá a recibir los mensajes antiguos.

3 DISEÑO

3.1 ARQUITECTURA DE LA APLICACIÓN

La aplicación sigue una arquitectura cliente-servidor. Esta arquitectura es un modelo de aplicación distribuida en el que las tareas se reparten entre los proveedores de recursos o servicios, llamados servidores, y los demandantes, llamados clientes.

3.2 UML

3.2.1 Diagrama de clases

3.2.1.1 Cliente

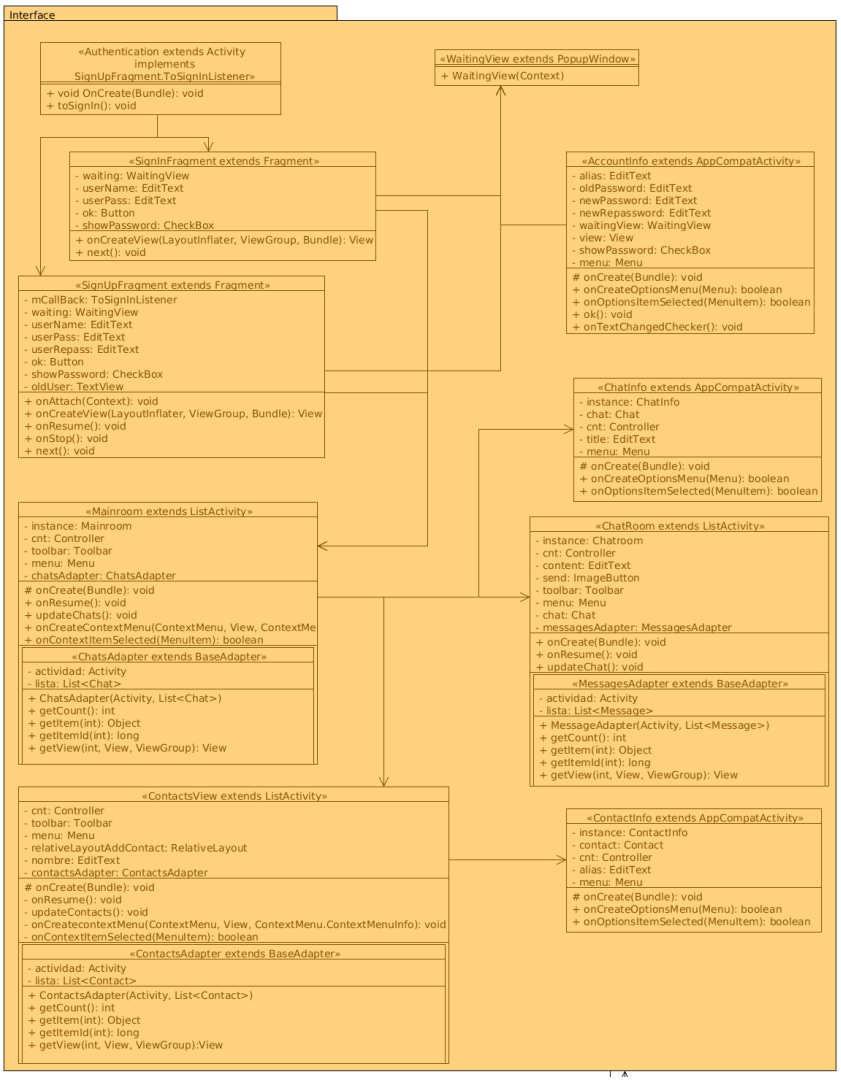

Ilustración 3-1 – Diagrama de clases del cliente: presentación

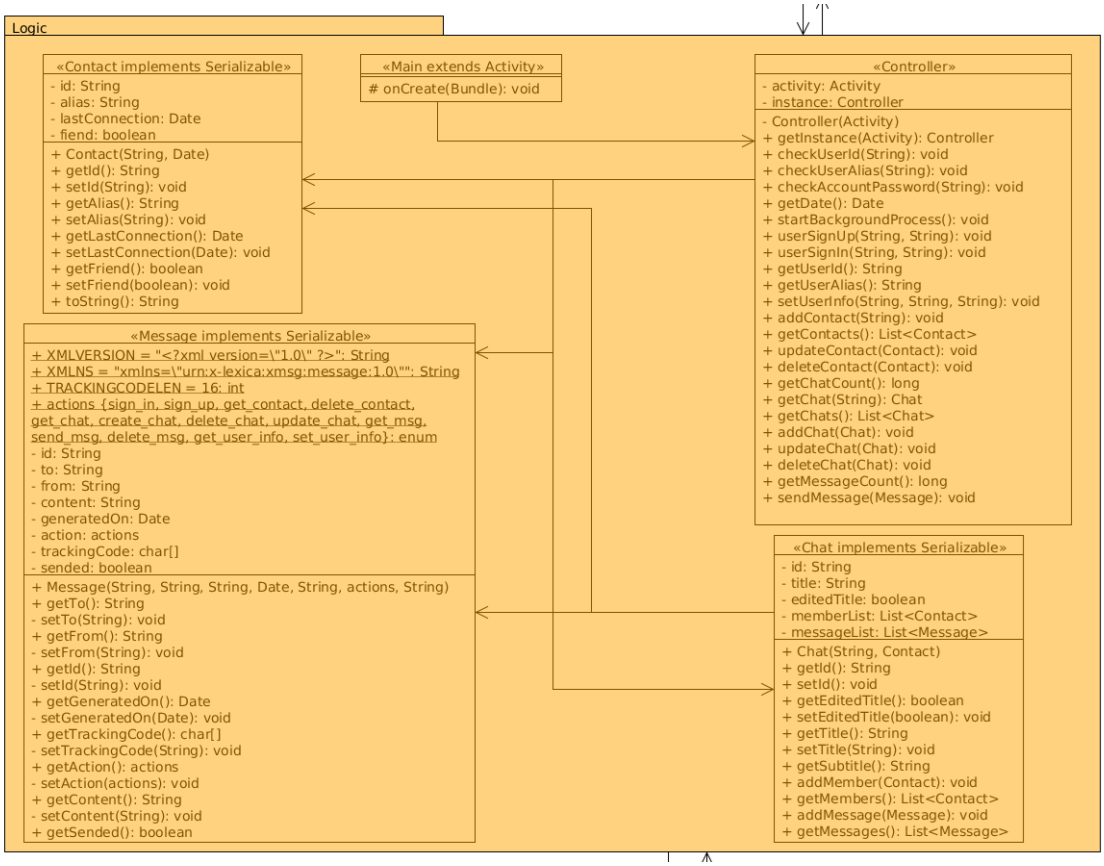

Ilustración 3-2 – Diagrama de clases del cliente: lógica

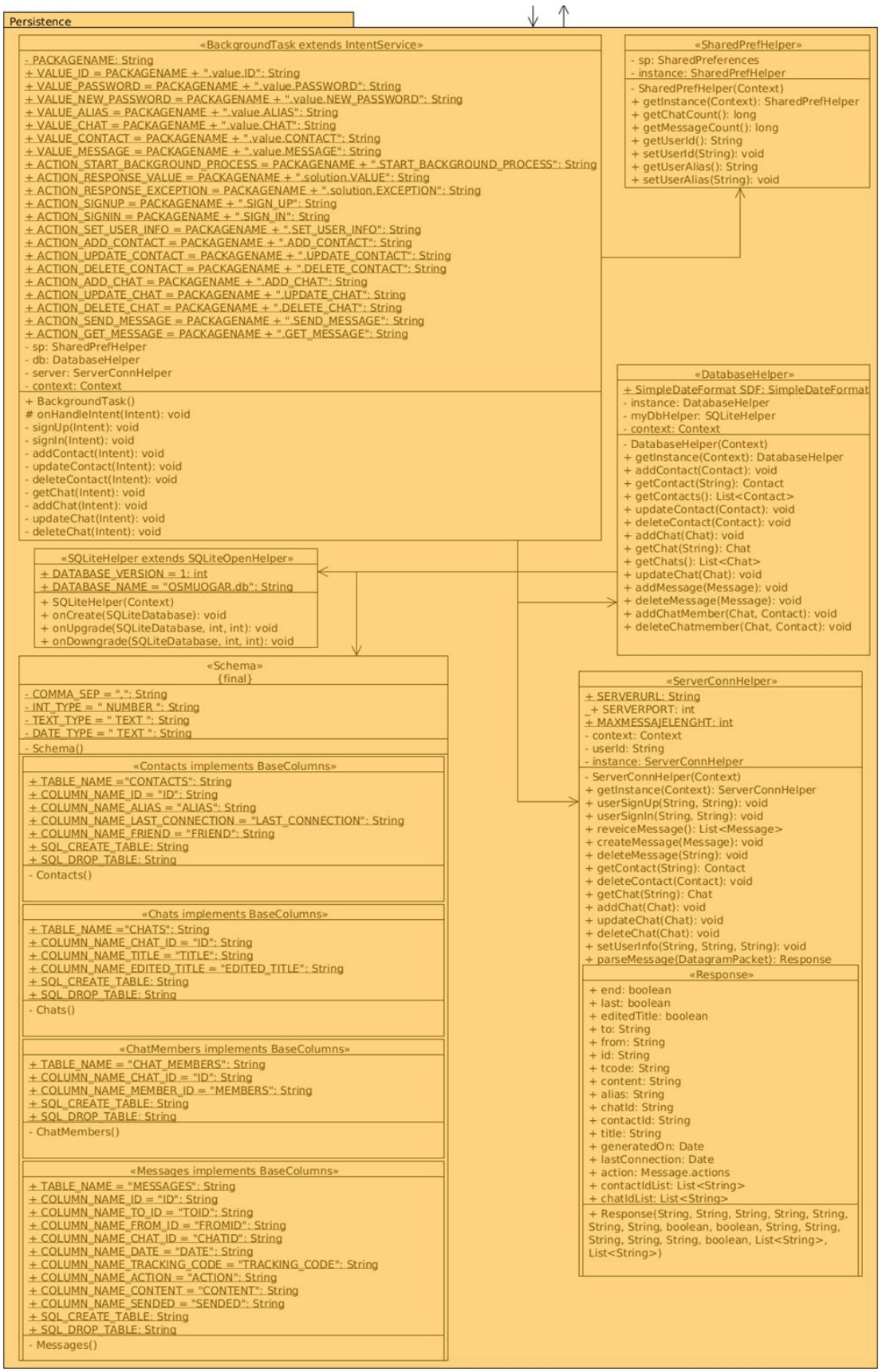

Ilustración 3-3 – Diagrama de clases del cliente: datos

3.2.1.2 Servidor

Ilustración 3-4 – Diagrama de clases del servidor

3.2.2 Diagramas de casos de uso

Los diagramas de casos de uso son una forma representa de una forma gráfica los casos de uso. Describen el comportamiento de la aplicación y enfatizan lo que debe suceder en el sistema modelado.

En nuestro sistema se distinguen dos actores:

Ilustración 3-5 – Diagrama de casos de uso

3.2.2.1 Usuario no identificado

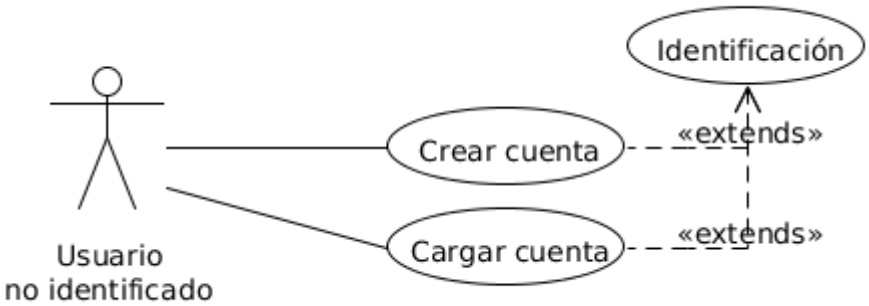

Ilustración 3-6 – Diagrama de casos de uso de usuario no identificado

Los usuarios no identificados únicamente pueden acceder al contenido de nuestra aplicación realizando una identificación.

3.2.2.2 Usuario identificado

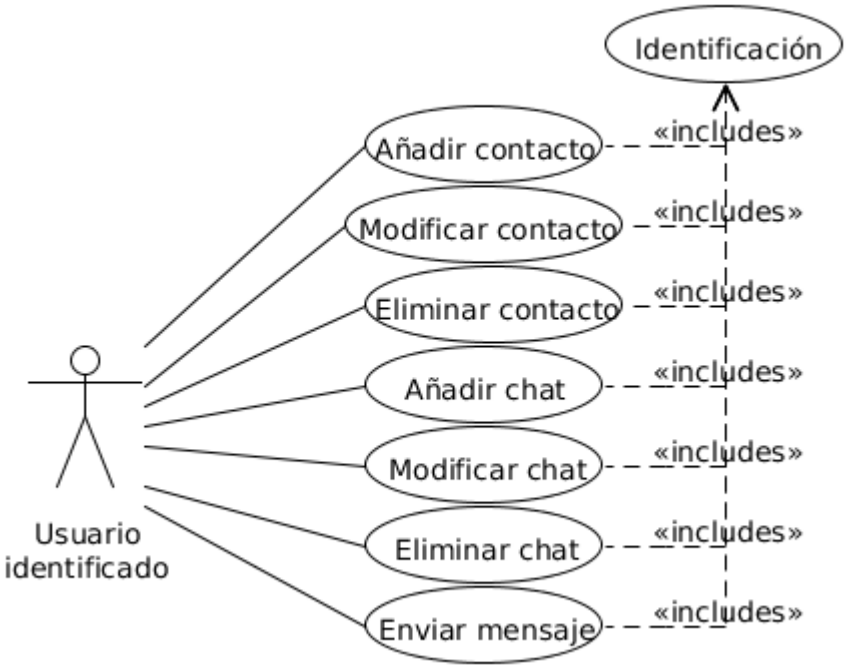

Ilustración 3-7 – Diagrama de casos de uso de usuario identificado

Los usuarios identificados pueden añadir, modificar y eliminar contactos, añadir, modificar y eliminar chats y enviar mensajes.

3.3 REFLEXIONES PREVIAS

3.3.1 UDP o TCP

El primer paso importante a decidir es si utilizar un *User Datagram Protocol* (UDP) o un *Transmission Control Protocol* (TCP) para la transmisión de los mensajes.

Para este caso concreto en el que estamos desarrollando una aplicación de mensajería, tenemos que tener en cuenta que pueden intentar comunicarse muchos usuarios a la vez y que mantener muchas conexiones TCP activas puede resultar un problema para el servidor y para la red en general. Es por esto que dadas las circunstancias parece más factible utilizar el protocolo UDP, ya que es un protocolo mucho más rápido y evitamos saturar demasiado la red.

Utilizar el protocolo UDP no nos asegura que el paquete llegue a su destino, pero hay otros sistemas para asegurar que el paquete llega a su destino por esto tendremos que implementar un *software* acorde a los problemas que nos plantea UDP.

3.3.2 Transmisión de datos

La transmisión de datos es la parte más importante de una aplicación de mensajería. Para simplificar esta explicación vamos a reducir el número de campos que se envía en los mensajes a “emisor”, “receptor” y “texto”.

En los mensajes tenemos que tener en cuenta poder identificar inequívocamente los tres campos que tendrán nuestros mensajes, ya que si no podemos identificar el campo receptor no podremos entregar el mensaje, si no podemos identificar el campo emisor no podemos advertir al usuario receptor de quién se está intentando comunicar, y

obviamente, si no podemos identificar el campo texto no podremos comunicárselo adecuadamente al usuario receptor.

Otro aspecto importante es el número de caracteres añadidos que vamos a utilizar para poder identificar los distintos campos del mensaje, ya que va a repercutir en el tamaño del paquete y en la cantidad de datos gastados por los dispositivos con tarifa de datos.

Por último, cabe destacar que el método que escojamos para realizar esto nos tiene que permitir ampliar la aplicación, mantenerla e intentar focalizar las partes complejas.

3.3.2.1 Método 1: JInterface

Existen varias librerías proporcionadas en la instalación de Erlang que facilitan la interacción con otros lenguajes. Entre ellas tenemos JInterface la cual permite la comunicación con Java, y otras como Erl_Interface que permiten la comunicación con C.

Este método implica que tanto el cliente como el servidor contienen *sockets* java. El *socket* Java del servidor se encargaría de la recepción de los mensajes y de su traducción a Erlang. Esto sería posible de dos modos: enviando caracteres de texto o enviando objetos Java.

Ventajas: toda la parte compleja de la aplicación estaría en el servidor, teniendo el cliente que preocuparse únicamente de comunicarse con el *socket* java del servidor. Los usuarios no tendrían ninguna limitación a la hora de escoger sus identificadores o a la hora de usar cualquier tipo de caracteres en los mensajes.

Desventajas: Como hemos comentado Erlang es una elección muy buena para servicios altamente disponibles, también es muy bueno en la concurrencia y en la creación de muchos hilos de forma simultánea. Tener un proceso Java en el servidor implica que hay un proceso del cual la estructura de Erlang para la recuperación de procesos no se puede hacer cargo de forma automática. Por otro lado, la creación de hilos en Java suele ser mucho más costosa que la creación de hilos en Erlang, por lo que puede ser un cuello de botella en la velocidad de nuestra aplicación.

Sin embargo no es fácil predecir si este método añadiría muchos caracteres innecesarios a nuestra aplicación, ya que de esto depende totalmente de la librería que permite enviar los objetos o cadenas de caracteres (dependiendo de cómo quisiéramos hacerlo).

3.3.2.2 Método 2: CSV

Transferencia de mensajes conteniendo el texto plano en formato *Comma Separated Values* (CSV). Podríamos obtener fácilmente el primer y el segundo parámetro, dejando todo a partir de la segunda coma para el mensaje de texto.

Ventajas: Este método nos asegura que los mensajes que se envían tienen pocos caracteres añadidos, lo que hace que tenga un peso bastante fuerte.

Desventajas: No hay una interfaz limpia para los programadores, haciendo difícil ampliar o mantener esta parte del código. Otro aspecto negativo es que nos obligaría a transmitir un mensaje por paquete enviado, ya que si quisiéramos enviar varios mensajes en un paquete no podríamos identificar el campo emisor y el campo receptor del segundo mensaje, debido a que no tenemos limitado el tamaño del texto del mensaje ni conocemos el número de comas que hay en el texto del mensaje. Finalmente este método tiene la limitación de que los identificadores de los usuarios no podrían contener comas.

3.3.2.3 Método 3: XML

Transferencia de un mensaje con texto formateado como *eXtensible Markup Language* (XML). Esta opción nos permitiría adoptar los estándares marcados por el *World Wide Web Consortium* (W3C).

Ventajas; Clara distinción entre los campos de los mensajes, pudiendo ser fácilmente mantenidos y ampliables. Permitiría enviar varios mensajes por paquete enviado. Sin caracteres restringidos ni para los identificadores de los usuarios ni para el texto.

Desventajas; el número de caracteres añadidos es mayor que en los otros dos métodos, ya que cada campo del mensaje tendrá como mínimo una etiqueta.

Hablando de tiempo de programación este es el más costoso, ya que este método implica que tanto en el servidor como en el receptor tendrá que haber un apartado específico que trate los mensajes. Sin embargo esto nos permitirá, en cierta medida, ampliar por separado las aplicaciones del cliente y del servidor.

3.3.3 Identificación de los usuarios

Utilizar el número de teléfono asegura que la mayoría de personas se identifican inequívocamente, puesto poseen únicamente un número de teléfono, cada persona tendría una cuenta y se le podría identificar fácilmente.

Este tipo de identificación facilita mucho la labor a la hora de administrar los usuarios en la base de datos, puesto que si podemos verificar el número de teléfono del terminal podemos asegurar que todos los números de teléfono de nuestra base de datos son distintos, por lo que ya tenemos una manera de distinguir a los usuarios inequívocamente en la base de datos.

Por otro lado, escoger el número de teléfono como identificador nos niega la posibilidad de crear un cliente para un dispositivo que no posea número de teléfono, como, por ejemplo, las tabletas Android, y nos impide o complica la posibilidad de implementar un futuro cliente para una plataforma web o una aplicación de escritorio. Puesto que es muy grande el número de dispositivos que están saliendo al mercado, no parece conveniente limitar la identificación de la aplicación a un número reducido de dispositivos que poseen número de teléfono.

Si conseguimos que la identificación de los usuarios no tenga limitación alguna o tenga las mínimas limitaciones posibles, podemos implementar en un futuro varios clientes dependiendo de las necesidades de los usuarios.

Que no liguemos algo de carácter personal como identificador implica que no podremos identificar a una persona con la cuenta, o nos sea mucho más difícil. De cualquier manera, relacionar a una persona con su cuenta no es algo obligatorio para nuestra aplicación, por lo que no tiene relevancia para el negocio que ofrece la aplicación asegurarnos de que una persona no tiene más de una cuenta o que es quien dice ser, dejando esta característica en un segundo plano.

Aunque no se impongan limitaciones directamente a la hora de escoger un identificador, esto no implica que no las haya, puesto que estaremos limitados por las características de las herramientas que traten los datos.

3.4 COMUNICACIÓN CLIENTE - SERVIDOR

3.4.1 Mensaje

Se utilizará el estándar de mensaje propuesto por el *World Wide Web Consortium* (W3C). Los campos que utilizaremos serán los obligatorios, el campo “tracking.code” y el campo “action”.

```
<message xmlns="urn:x-lexica:xmsg:message:1.0"
        to="{uri}"
        from="{uri}"
        id="{string}"
        generated.on="{datetime}"
        reply.to="{uri}"
        originator="{uri}"
        originator.id="{string}"
        priority="{lowest|low|normal|high|highest}"
        expires="{datetime}"
        tracking.code="{string}"
        action="{string}"
        manifest="{uri}"
        receipt.required="{boolean}"
        for.receipt="{string}"
        >

        <hop/> ...
        <property/> ...
        <document>...</document> ... | <failure/> | <receipt/>

</message>
```

Atributo	Descripción	Requerido
from	La URI que identifica al emisor del mensaje.	Si
to	La URI que identifica al receptor del mensaje.	Si
reply.to	La URI a usar en la respuesta para el atributo "to".	No
originator	La URI que identifica quien envió el mensaje al actual emisor del mensaje.	No
originator.id	El identificador del mensaje inicial del originador.	No
priority	Una prioridad entre *lowest*, *low*, *normal*, *high* o *highest*, que por defecto contiene "normal."	No
expires	Una fecha y tiempo en que el mensaje expira.	No
id	Un identificador del mensaje único para el emisor.	Si
generated.on	La fecha y hora en que el mensaje fue generado.	Si
tracking.code	Código de rastreo específico de una aplicación.	No
action	Un sujeto específico para una aplicación.	No

Atributo	Descripción	Requerido
manifest	Un URI que identifica una descripción del contenido del mensaje	No
receipt.required	Un indicador de que se requiere una respuesta.	No
for.receipt	El valor de una respuesta previa por el cual este mensaje es una contestación.	No

3.4.2 Fallos

Un elemento de fallo cifra un fallo a la hora de tratar un mensaje. Típicamente este elemento es usado para enviar una respuesta a un mensaje recibido que no pudo ser procesado.

```
<failure on="{timeInstant}" reason.code="{string}">...</failure>
```

El contenido de este elemento es un mensaje textual asociado al fallo. El tiempo en el que el procesamiento ha fallado es transmitido en el atributo “on”, el cual es requerido.

Opcionalmente, el Sistema emisor puede codificar un código para establecer la razón del fallo en el atributo “reason.code”.

3.4.3 Recepciones

Las recepciones son retornadas como réplica al mensaje cuando el atributo “receipt.required” es cierto. Esta recepción puede ser usada para verificar que un mensaje ha sido recibido de forma asíncrona, el cual puede ser comprobado con el atributo “message.tracking.code” en la recepción.

```
<receipt message.tracking.code="{string}" timestamp="{timeInstant}"/>
```

Cuando la recepción es generada, el instante de tiempo es guardado en el atributo “timestamp”. El código de rastreo del mensaje es el valor del atributo “message.tracking.code”. Ambos atributos son requeridos.

3.4.4 Contenido dependiendo de la acción

sign_up, sign_in:

```
<password>password</password>
```

get_contact, delete_contact:

```
<id>contactId</id>
```

get_chat:

```
<id>chatId</id>
```

create_chat:

```
<id>chatId</id>
<title>chatTitle</title>
<edited>chatEditedTitle</edited>
<user>chatMember</user>
```

update_chat:

<id>chatId</id>

<title>chatTitle</title>

<edited>chatEditedTitle</edited>

delete_chat:

<id>chatId</id>

set_user_info:

<alias>userAlias</alias>

<old_password>oldPassword</old_password>

<new_password>newPassword</new_password>

get_msg:

#nada

send_msg:

#texto del mensaje

delete_msg:

<id>messageId</id>

4 IMPLEMENTACIÓN

4.1 ARQUITECTURA DEL CLIENTE

Para la aplicación cliente se utilizará la programación por capas. La programación por capas es una arquitectura cliente-servidor en el que el objetivo primordial es la separación de la lógica de negocios de la lógica de diseño.

4.1.1 Nivel de presentación

El nivel de presentación se encarga de la representación de la información de forma visual, de manera que, aunque distintos equipos puedan tener diferentes representaciones internas, los datos llegan de manera reconocible. Se definen desde los colores y las fuentes hasta un diseño de las interfaces y su contenido.

4.1.1.1 Colores

Haciendo un breve estudio de los colores podemos identificar qué colores deberíamos de utilizar para la aplicación.

Ilustración 4-1 - Guía de color en marketing

En nuestro caso tenemos que tener en cuenta que hay muchas aplicaciones de mensajería y hay que escoger un color con el que los usuarios nos puedan identificar.

Debido a ciertos estudios acerca de los estilos publicados por Google, se recomienda utilizar dos colores, los cuales van a ser usados uno como principal y otro como acento para los mensajes o ventanas que requieran nuestra atención.

Color	Significado	Su uso aporta	El exceso produce
BLANCO	Pureza, inocencia, optimismo	Purifica la mente a los más altos niveles	---
LAVANDA	Equilibrio	Ayuda a la curación espiritual	Cansado y desorientado
PLATA	Paz, tenacidad	Quita dolencias y enfermedades	---
GRIS	Estabilidad	Inspira la creatividad Simboliza el éxito	---
AMARILLO	Inteligencia, alentador, tibieza, precaución, innovación	Ayuda a la estimulación mental Aclara una mente confusa	Produce agotamiento Genera demasiada actividad mental
ORO	Fortaleza	Fortalece el cuerpo y el espíritu	Demasiado fuerte para muchas personas
NARANJA	Energía	Tiene un agradable efecto de tibieza Aumenta la inmunidad y la potencia	Aumenta la ansiedad
ROJO	Energía, vitalidad, poder, fuerza, apasionamiento, valor, agresividad, impulsivo	Usado para intensificar el metabolismo del cuerpo con efervescencia y apasionamiento Ayuda a superar la depresión	Ansiedad de aumentos, agitación, tensión
PÚRPURA	Serenidad	Útil para problemas mentales y nerviosos	Pensamientos negativos
AZUL	Verdad, serenidad, armonía, fidelidad, sinceridad, responsabilidad	Tranquiliza la mente Disipa temores	Depresión, aflicción, pesadumbre
AÑIL	Verdad	Ayuda a despejar el camino a la consciencia del yo espiritual	Dolor de cabeza
VERDE	Ecuanimidad inexperta, acaudalado, celos, moderado, equilibrado, tradicional	Útil para el agotamiento nervioso Equilibra emociones Revitaliza el espíritu Estimula a sentir compasión	Crea energía negativa
NEGRO	Silencio, elegancia, poder	Paz. Silencio	Distante, intimidatorio

Ilustración 4-2 – Significado colores

Teniendo en cuenta las cualidades y los excesos, se han escogido el color naranja como principal y el color azul como color de acentuación.

Naranja500#FF9800	Azul500#2196F3
50#FFF3E0	50#E3F2FD
100#FFE0B2	100#BBDEFB
200#FFCC80	200#90CAF9
300#FFB74D	300#64B5F6
400#FFA726	400#42A5F5
500#FF9800	500#2196F3
600#FB8C00	600#1E88E5
700#F57C00	700#1976D2
800#EF6C00	800#1565C0
900#E65100	900#0D47A1
A100#FFD180	A100#82B1FF
A200#FFAB40	A200#448AFF
A400#FF9100	A400#2979FF
A700	A700

De la gama del color principal debemos de escoger 3, mientras que de la gama del color que será utilizado como acento debemos escoger uno.

Lo recomendado es utilizar los colores de la primera fila, como colores principales, y utilizar cualquiera de los 3 colores de la última fila como color de acentuación.

Color principal 1: Naranja500#FF9800

Color principal 2: 700#F57C00

Color principal 3: Blanco#FFFFFF

Color acento: A200#448AFF

4.1.1.2 Fuentes

Pese a que pueda parecer irrelevante, ciertos estudios demuestran que las fuentes y los colores de las fuentes deben de ser precisos.

Las letras serán más grandes e intensas cuanto más importante sea el texto, a medida que pierda importancia se utilizarán tamaños más pequeños o colores menos intensos.

Fuente utilizada: Roboto.

Tamaños:

```
<dimen name="font_extra_small">12sp</dimen>
<dimen name="font_small">14sp</dimen>
<dimen name="font_medium">16sp</dimen>
<dimen name="font_big">20sp</dimen>
<dimen name="font_extra_big">34sp</dimen>
```

Colores:

```
<color name="primary_text">#DE000000</color><!-- 87% -->
<color name="secondary_text">#8A000000</color><!-- 54% -->
<color name="hint_text">#61000000</color><!-- 38% -->
<color name="accentuated_background_text">#ffffffff</color>
    <!-- 100% ->
```

4.1.1.3 Interfaces

4.1.1.3.1 Registro

Primera interfaz que visualiza el usuario. Esta interfaz está dedicada para registrarse en nuestra aplicación.

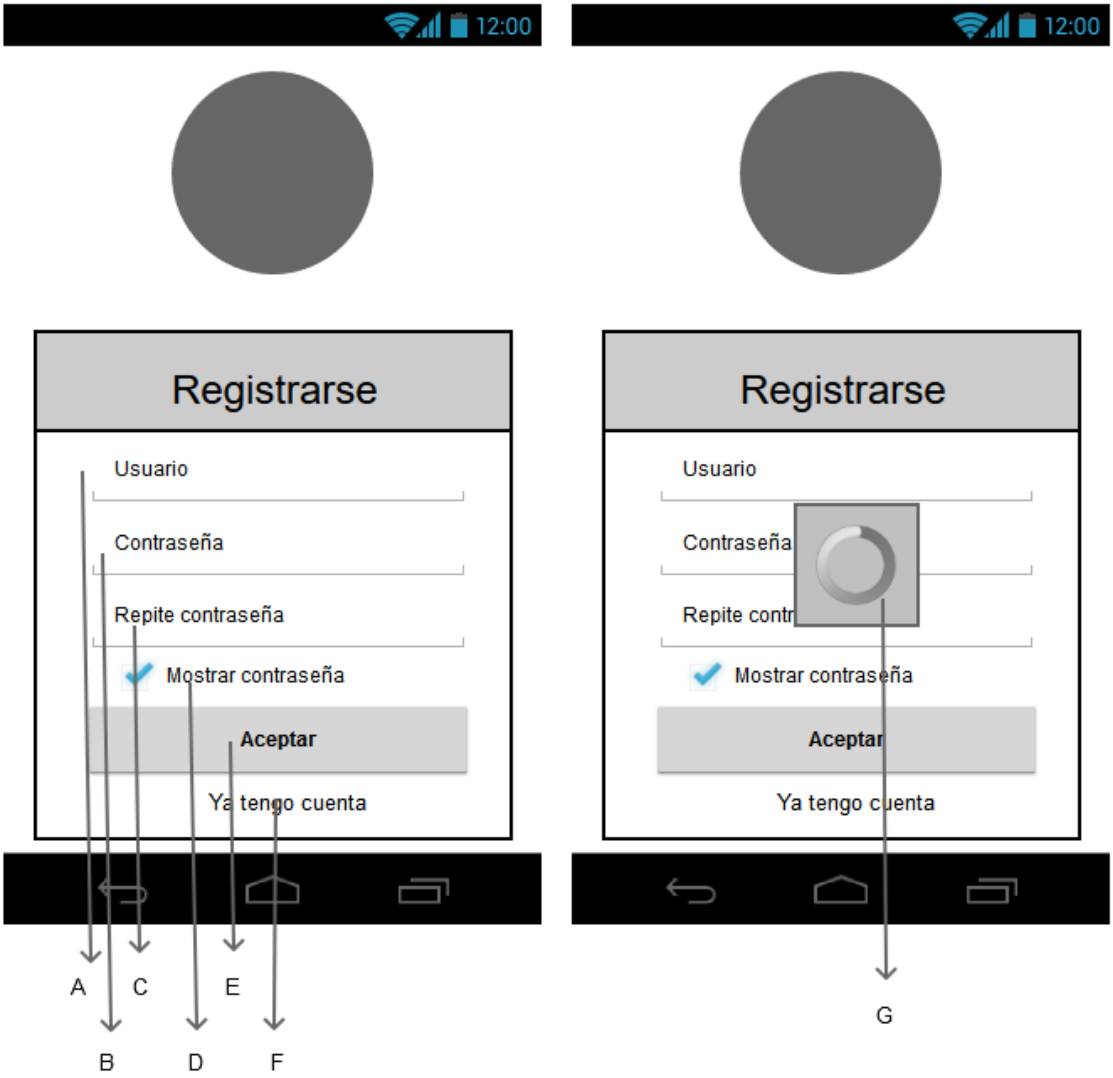

Ilustración 4-3 – Interfaz registro

A. Nombre de usuario. El tamaño mínimo es de 2 caracteres.
B. Contraseña. El tamaño mínimo es de 4 caracteres.
C. Repetición de la contraseña.
D. Si está marcada muestra la contraseña y la repetición de la contraseña.
E. Se empieza un proceso de comprobación. Las contraseñas se ocultan. El nombre del usuario y la contraseña deben de contener un mínimo de caracteres, y ambas contraseñas deben de coincidir. Si estas pre condiciones son correctas, se crea una interfaz de espera para el usuario.
   Cuando la información se envía al servidor se comprueba que el usuario no existe ya en el sistema.
   Hay tres posibles opciones para que desaparezca la interfaz E.
   1. La creación de la cuenta ha sido satisfactoria. Por lo que pasamos a la interfaz principal.

2. La creación de la cuenta ha fallado porque ya existe el usuario en el sistema. En este caso avisamos al usuario con un mensaje en la pantalla.
3. La conexión con el servidor ha fallado. Avisamos al usuario con un mensaje en la pantalla.

F. Enlace a la interfaz de inicio de sesión.
G. Interfaz de espera.

4.1.1.3.2 Inicio de sesión

Esta interfaz es para que aquellos que ya tienen cuenta puedan volver a acceder cuando instalan la aplicación en otro dispositivo.

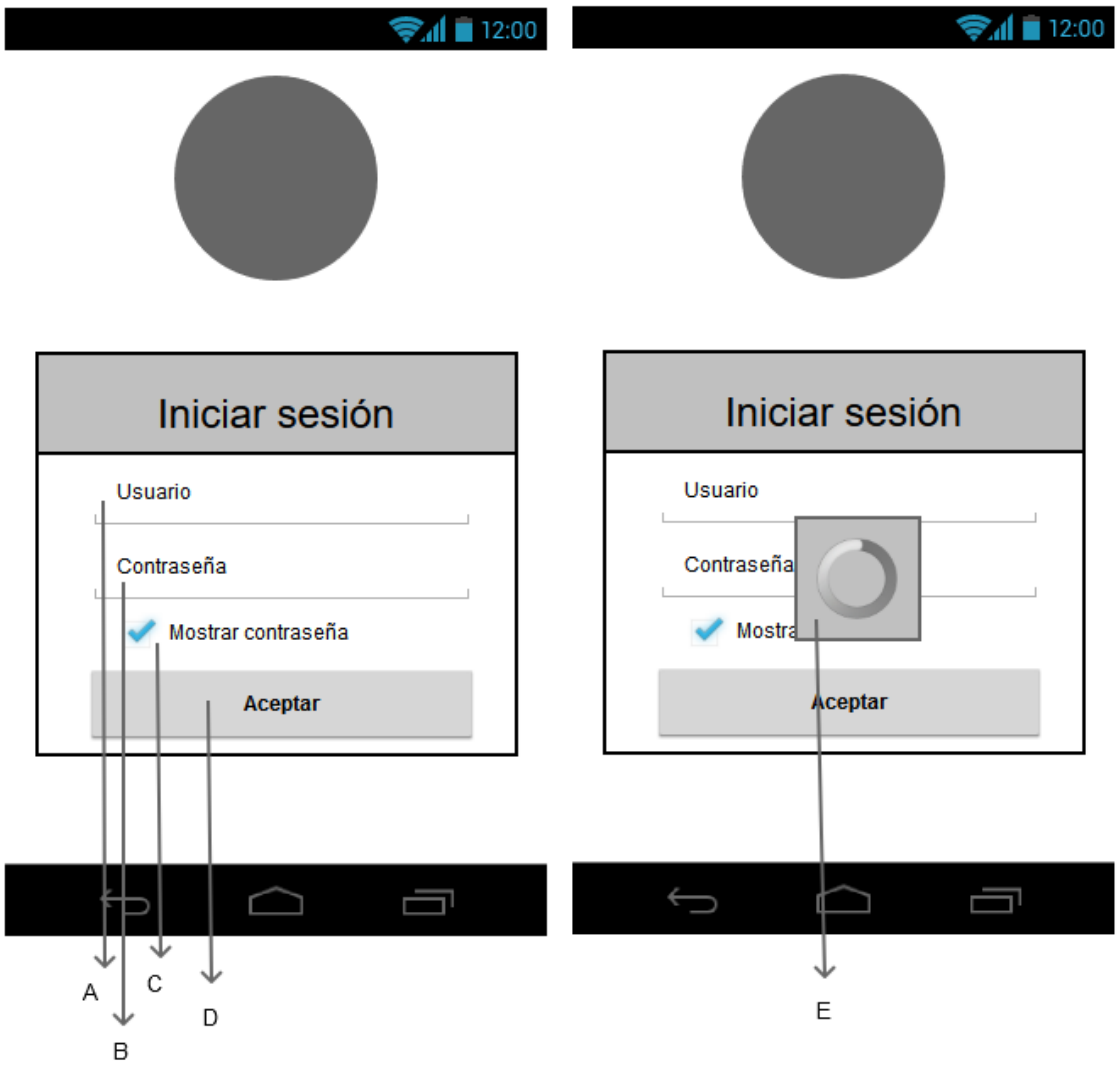

Ilustración 4-4 – Interfaz inicio de sesión

A. Nombre de usuario.
B. Contraseña.
C. Si está marcada muestra la contraseña y la repetición de la contraseña.
D. Se empieza un proceso de comprobación. La contraseña se oculta. Se crea una interfaz de espera para el usuario.
E. Cuando la información se envía al servidor se comprueba que el usuario existe en el sistema.
   Hay tres posibles opciones para que desaparezca la interfaz E:

- La cuenta se ha cargado exitosamente. Por lo que pasamos a la interfaz principal.
- La cuenta no se ha podido cargar por un fallo en el nombre de usuario o en la contraseña. En este caso avisamos al usuario con un mensaje en la pantalla.
- La conexión con el servidor ha fallado. Avisamos al usuario con un mensaje en la pantalla.

F. Interfaz de espera.

4.1.1.3.3 Principal

Interfaz principal de la aplicación. En ella se almacenan los chats activos de los usuarios.

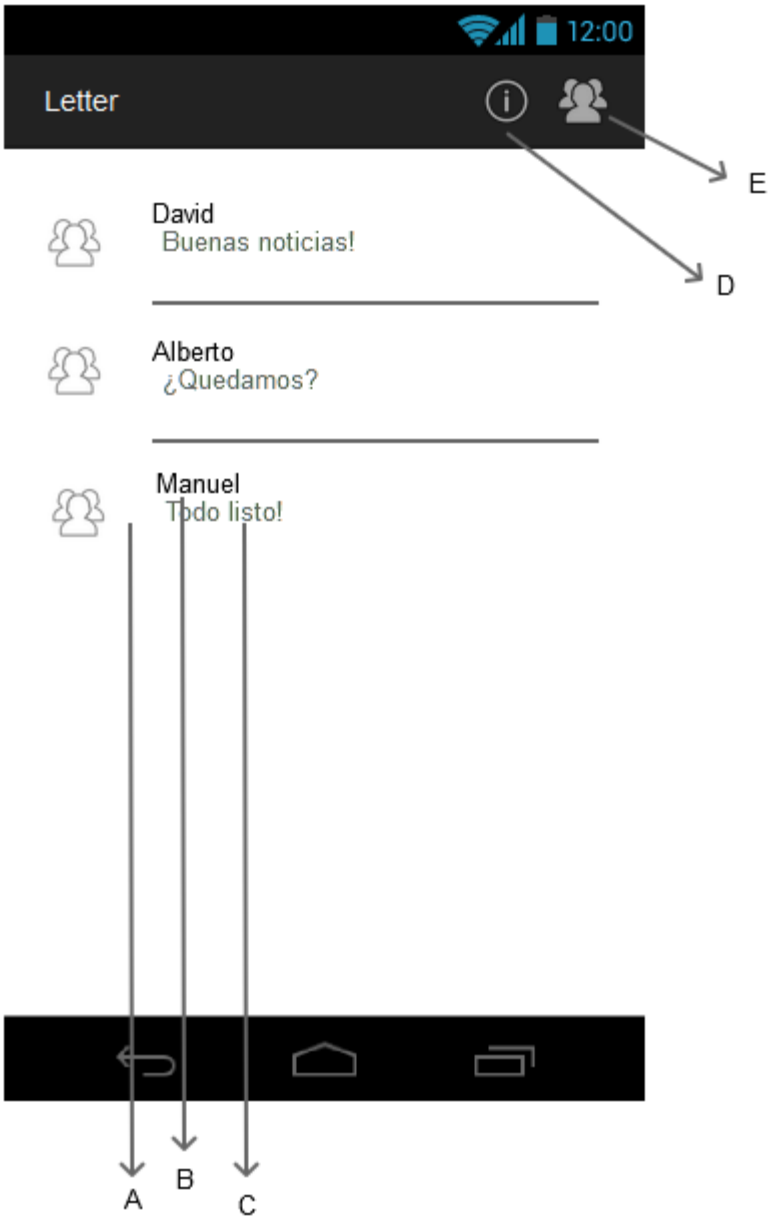

Ilustración 4-5 – Interfaz principal

A. Chat de usuario. Al pinchar sobre este icono accedemos a la interfaz de chat.
B. Nombre del chat.
C. Último mensaje enviado en el chat.
D. Enlace a la interfaz información de la cuenta.
E. Enlace a la interfaz contactos.

4.1.1.3.4 Información de la cuenta

En esta interfaz se muestra y se permite la modificación de la información relativa a la cuenta.

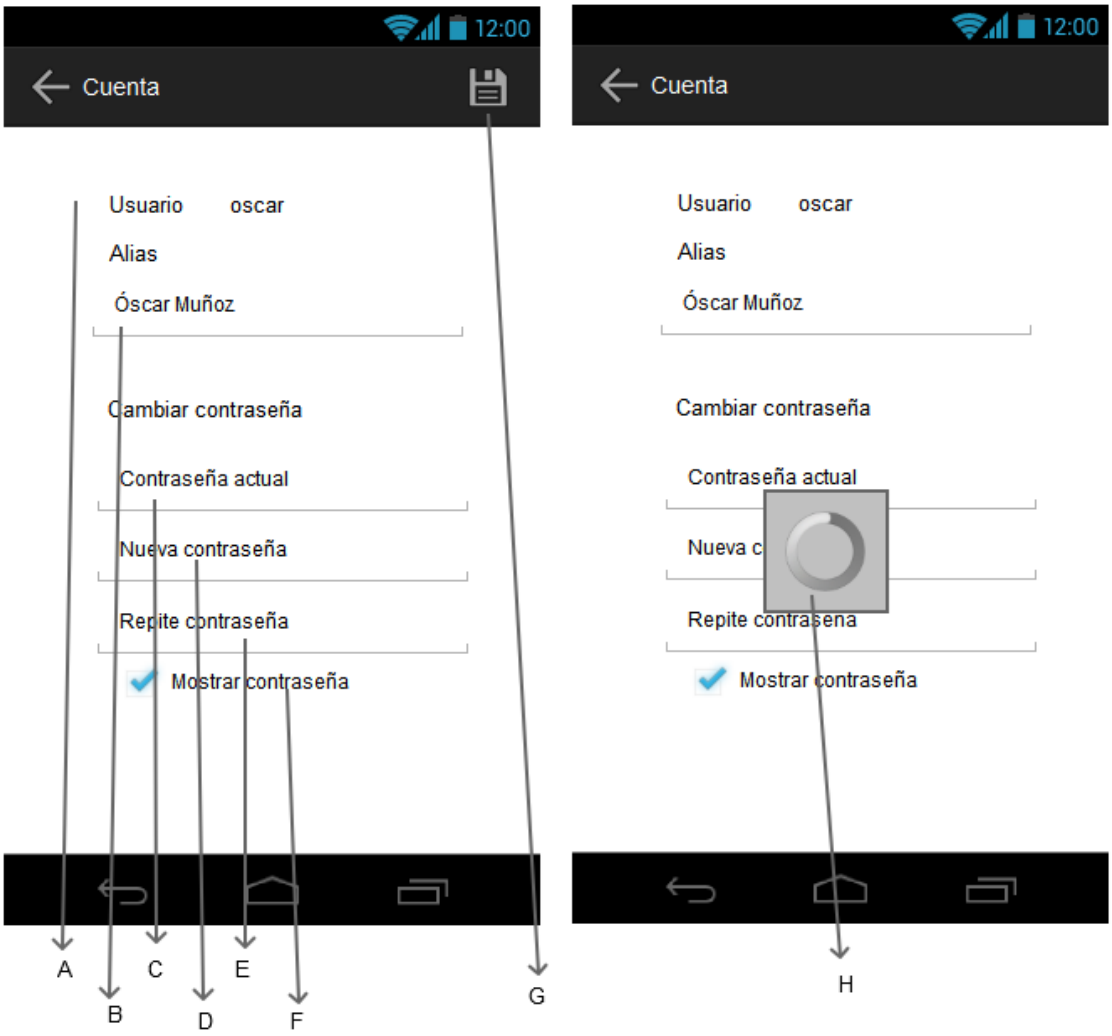

Ilustración 4-6 – Interfaz información de la cuenta

A. Nombre de usuario. No puede ser modificado, ya que es el identificador que utiliza el sistema para poder identificar al usuario de forma inequívoca.
B. Alias. El alias un nombre amistoso que se muestra a los usuarios en sustitución del identificador de usuario.
C. Contraseña actual
D. Nueva contraseña.
E. Repetición de la nueva contraseña.
F. Si está marcada muestra la contraseña actual, la nueva contraseña y la repetición de la contraseña.
G. Icono de guardado. Este icono únicamente aparece si todas las precondiciones para que los datos sean válidos se cumplen. Dichas precondiciones son:
   - El alias debe contener un mínimo de 2 caracteres.
   - Las contraseñas se han modificado todas o no se ha modificado ninguna.

   Cuando el botón se pulsa aparece una interfaz de espera.
H. Interfaz de espera. Para que esta interfaz desaparezca se pueden dar tres motivos:
   - Los cambios han sido guardados satisfactoriamente.

- Los cambios no han podido ser guardados debido a un fallo en la comprobación de la contraseña actual. Un mensaje de texto es mostrado advirtiendo al usuario
- No hay conexión con el servidor. Un mensaje de texto es mostrado advirtiendo al usuario.

4.1.1.3.5 Contactos

Esta interfaz muestra todos los contactos que tenemos en la aplicación.

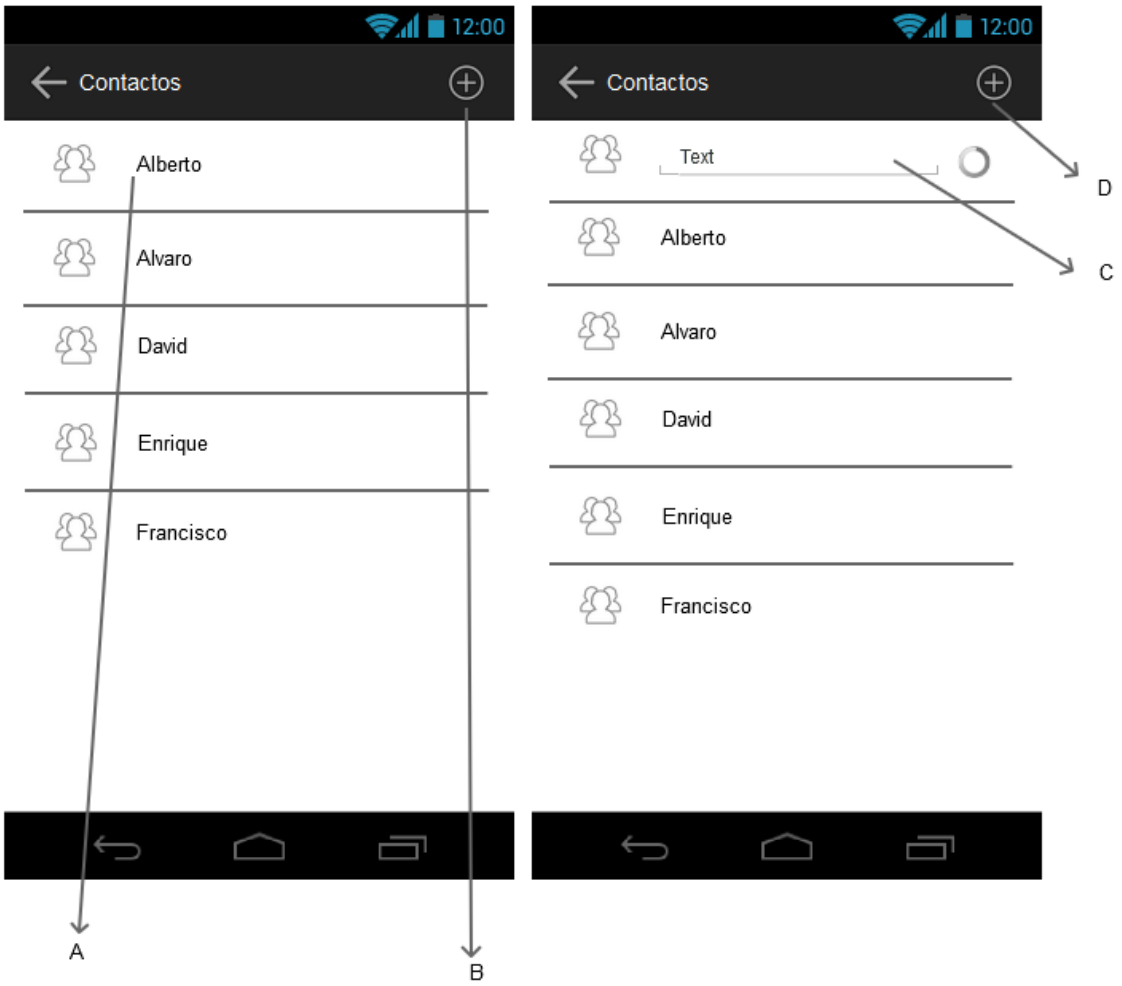

Ilustración 4-7 – Interfaz contactos

A. Contacto. Al mantener sobre el icono del contacto será desplegado un menú contextual. En el menú se puede crear un chat o acceder a la información del contacto.
B. Botón añadir contacto. Al pulsar sobre este botón aparece la interfaz añadir contacto.
C. Añadir contacto. Aquí añadimos el nombre de la cuenta de nuestro contacto. Cuando el nombre es válido aparece el icono para la confirmación.
D. Confirmar añadir contacto. Al pulsar el botón aparece un icono de espera al lado de añadir contacto. Hay dos posibilidades para que desaparezca dicho icono:
    - El contacto existe. La interfaz de añadir contacto desaparece.
    - El contacto no existe. Se notifica al usuario con un mensaje de texto en pantalla. La interfaz de añadir contacto desaparece.

4.1.1.3.6 Información del contacto

Esta interfaz es muestra la información que tenemos de un contacto.

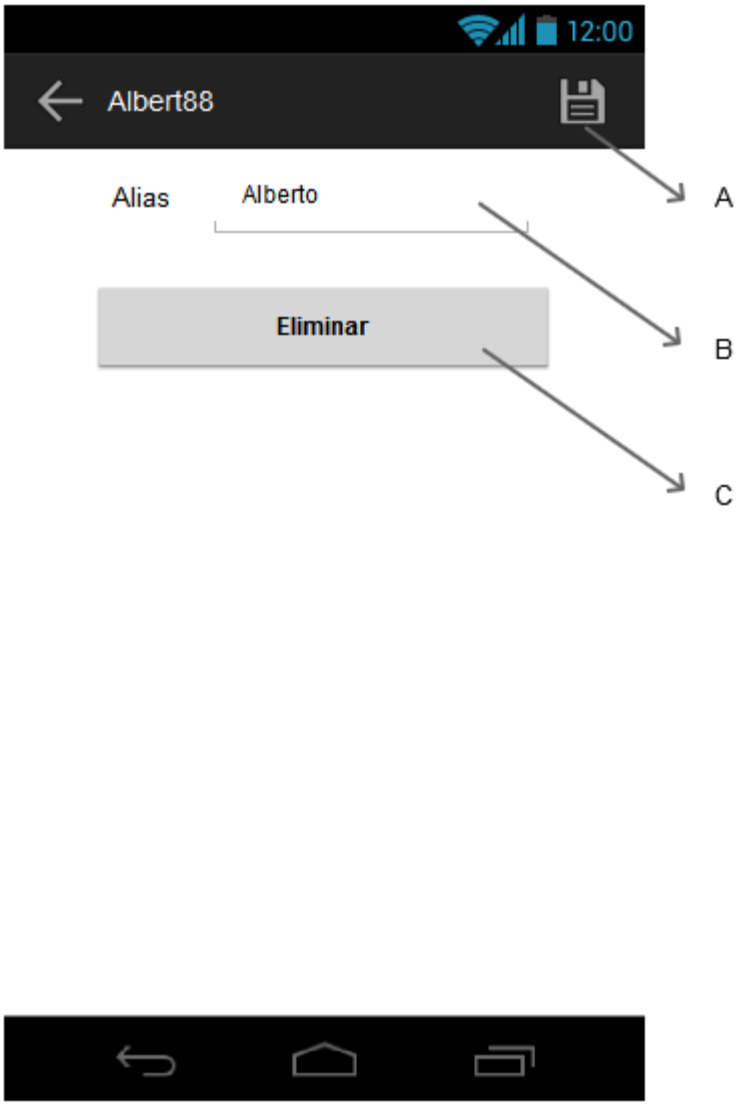

Ilustración 4-8 – Interfaz información del contacto

A. Icono de guardar cambios. El icono únicamente aparece cuando los cambios son válidos.
B. Alias del contacto. El alias sustituye de forma visual el nombre de usuario del contacto.
C. Se elimina el usuario de la lista de amigos.

4.1.1.3.7 Chat

Interfaz de chat. En esta interfaz es donde podemos enviar y recibir mensajes de otros usuarios.

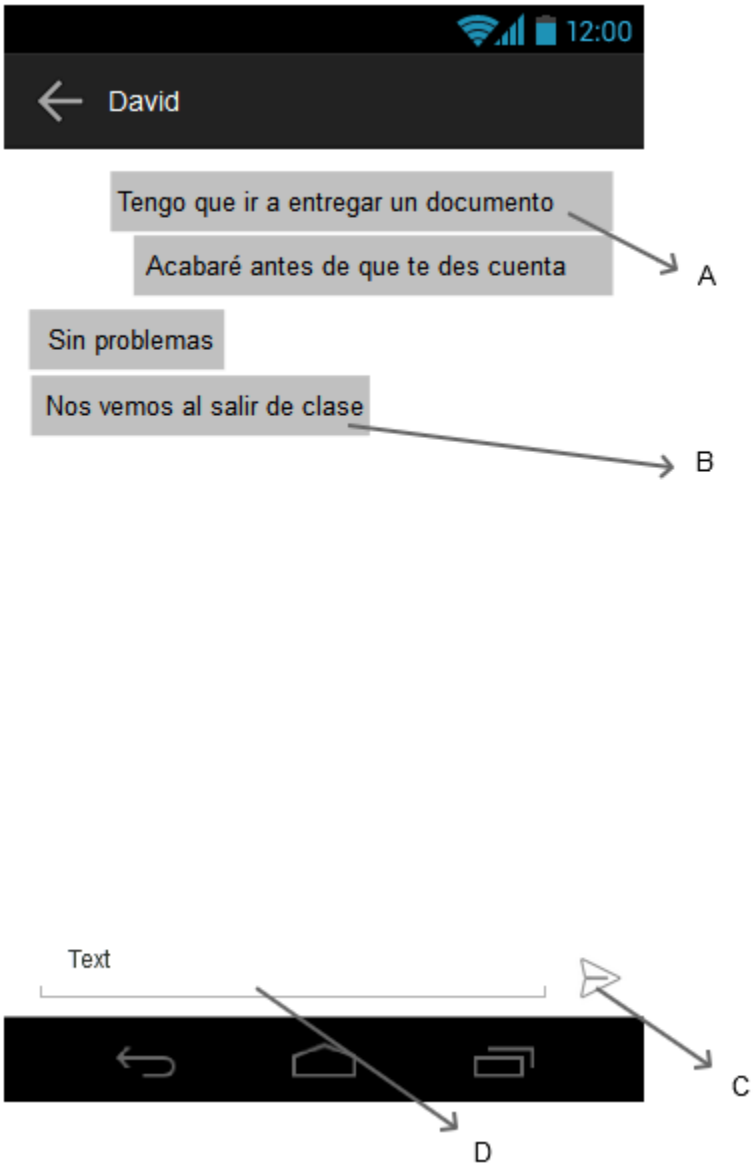

Ilustración 4-9 – Interfaz chat

A. Mensajes que envía el usuario local. Aparecen en el lado derecho de la pantalla
B. Mensajes recibidos. Aparecen en el lado izquierdo de la pantalla.
C. Botón de enviar. El botón únicamente aparece cuando el texto a enviar es válido. Dicho texto es válido cuando su tamaño es mayor a uno.
D. Texto a enviar.

4.1.2 Nivel de aplicación

El nivel de aplicación ofrece a las aplicaciones la posibilidad de acceder a los servicios de las demás capas y define los protocolos que utilizan las aplicaciones.

- Main: Se encarga de la ejecución inicial del programa.
- Controlador: Controla las conexiones entre los distintos niveles.
- Chat: contiene el id del chat, el título, los mensajes y los miembros del chat.
- Contacto: contiene el id del contacto, el alias y la última conexión.
- Mensaje: contiene el id del mensaje, los campos to, from, la fecha en la que se crea el mensaje, un código para rastrear el mensaje y el contenido del mensaje.

4.1.3 Nivel de persistencia

El nivel de persistencia se encarga de preservar la información de un objeto de forma permanente, pero a su vez también se refiere a poder recuperar la información del mismo para que pueda ser nuevamente utilizado.

El cliente tiene tres recursos de persistencia:

- SharedPreferences: se de almacenamiento de datos clave-valor. Todas las aplicaciones de Android pueden acceder a este almacenamiento con un coste muy bajo. Destaca por un peso muy pequeño y una velocidad considerable. Se caracteriza por almacenar datos de tipo primitivo.
- SQLite: Sqlite es la base de datos que utilizaremos para guardar toda la información que crea el usuario y toda aquella que recibiremos de nuestra aplicación servidor.
- Conexión con el servidor: La conexión con el servidor se considera en el nivel de persistencia, ya que lo que haremos con esta conexión es almacenar los datos en un lugar remoto donde podrán ser consultados con posterioridad.

4.1.4 Implementación detallada

A continuación se muestra la descripción detallada de la implementación del proyecto, con ejemplos de código.

4.1.4.1 Capa de presentación

Para la capa de presentación del cliente se utiliza xml para el diseño y una clase java para aplicar la funcionalidad.

Ilustración 4-10 – Interfaz crear cuenta

Ilustración 4-11 – Interfaz cargar cuenta

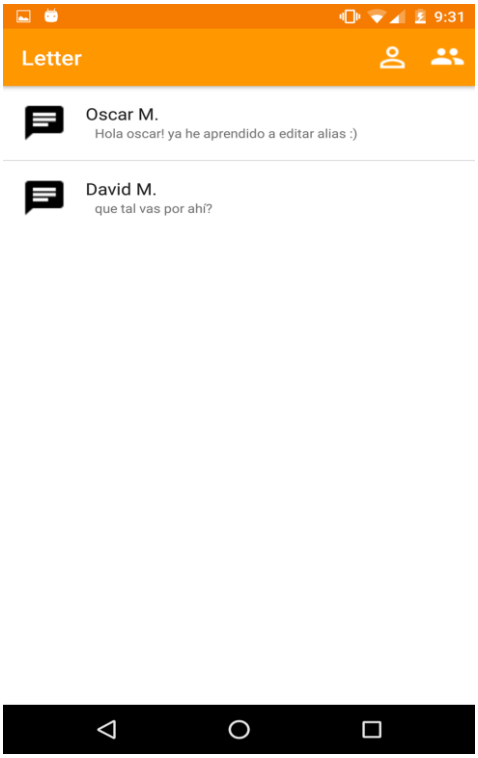

Ilustración 4-12 – Interfaz principal

Ilustración 4-13 – Interfaz contactos

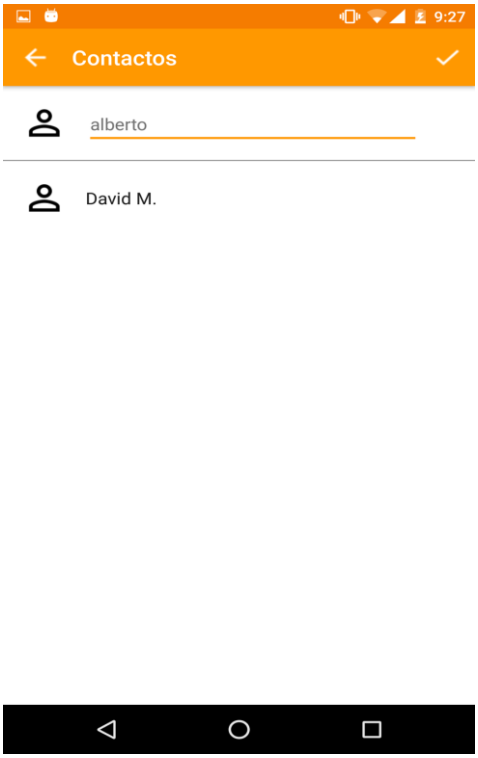

Ilustración 4-14 – Interfaz añadir contacto

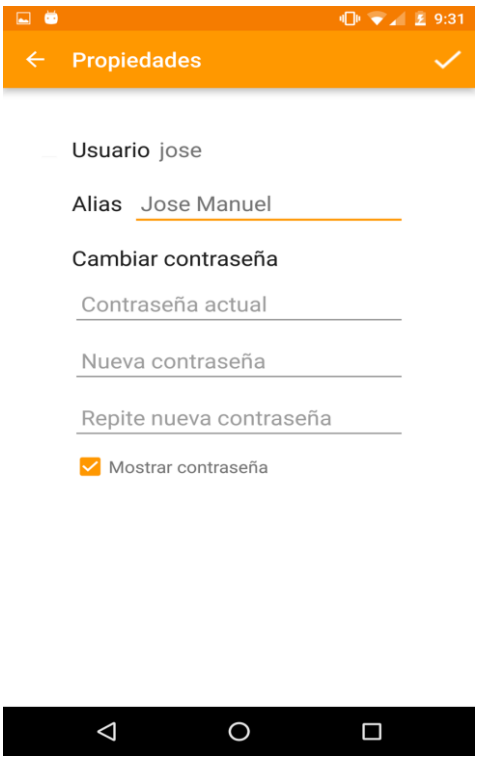

Ilustración 4-15 – Interfaz información de cuenta

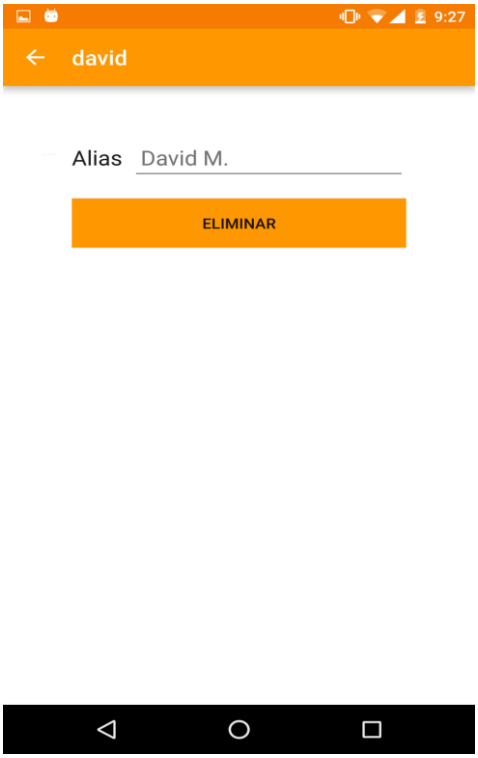

Ilustración 4-16 – Interfaz información de contacto

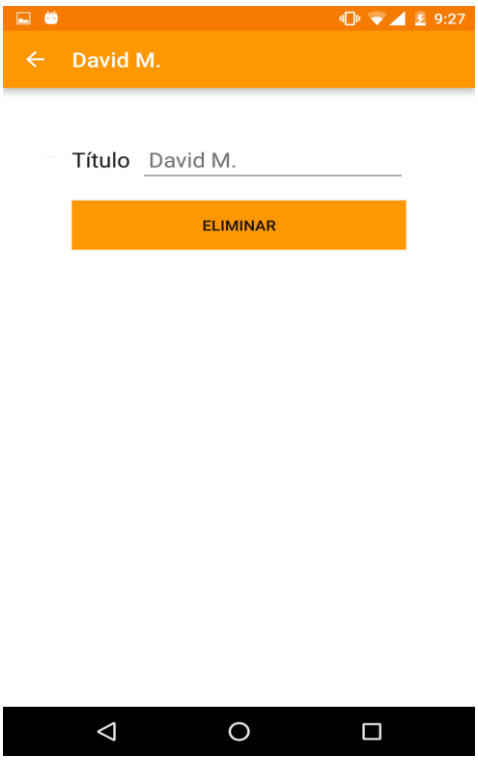

Ilustración 4-17 – Interfaz información de chat

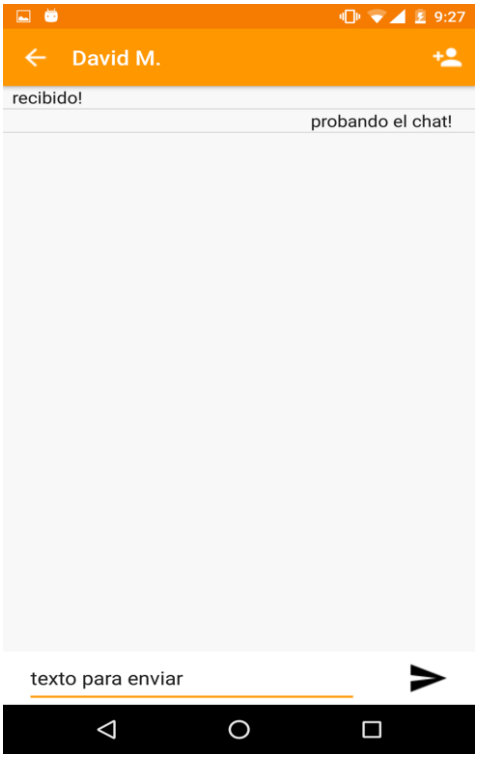

Ilustración 4-18 – Interfaz chat

4.1.4.2 Capa de aplicación

4.1.4.2.1 Transmisión/*Broadcast*

Con los Broadcast podemos comunicar los procesos en segundo plano con los procesos en primer plano.

Este es el código en el proceso de segundo plano que lanza una transmisión. En este caso se trata del *signin* del usuario.

```
try {
        String userId = intent.getStringExtra(VALUE_ID);
        server.userSignUp(userId, intent.getStringExtra(VALUE_PASSWORD));
        sp.setUserId(userId);
        sp.setUserAlias(userId);
} catch (NoServerConnectionException e) {
        intent.putExtra(ACTION_RESPONSE_EXCEPTION,
                new NoServerConnectionException());
} catch (AccountCreationFailureException e) {
        intent.putExtra(ACTION_RESPONSE_EXCEPTION,
                new AccountCreationFailureException());
}
LocalBroadcastManager.getInstance(context).sendBroadcast(intent);
```

Como podemos comprobar, dependiendo del resultado de la transacción del servidor unimos un extra u otro al *intent* que enviaremos mediante el *broadcast*.

```
BroadcastReceiver receiver = new BroadcastReceiver() {
        @Override
        public void onReceive(final Context context, Intent intent) {
                final Exception solution_exceptions =(Exception)
                intent.getSerializableExtra
                (BackgroundTask.ACTION_RESPONSE_EXCEPTION);
                if (solution_exceptions != null) {//Si hay excepciones
                        // Notificamos al usuario el problema
                }
                updateContacts();
                /* Código no relevante */
        }
};
```

```
LocalBroadcastManager.getInstance(getApplicationContext()).registerReceiver(receiver, new IntentFilter(BackgroundTask.ACTION_ADD_CONTACT));

LocalBroadcastManager.getInstance(getApplicationContext()).registerReceiver(receiver, new IntentFilter(BackgroundTask.ACTION_DELETE_CONTACT));

LocalBroadcastManager.getInstance(getApplicationContext()).registerReceiver(receiver, new IntentFilter(BackgroundTask.ACTION_UPDATE_CONTACT));
```

En este caso el *BroadcastReceiver* es para los contactos. Además registramos el *BroadcastReceiver* para varias acciones, en este caso serán añadir contacto, eliminar contacto y actualizar contacto, por lo que cuando se realice cualquier acción con los contactos, esta vista estará actualizada al instante.

4.1.4.2.2 Listado/ListView

Las *ListView* son utilizadas para hacer una lista de los objetos. En este caso para la aplicación las hemos utilizado para los contactos, para los chats y para los mensajes.

Las *ListView* son importantes porque permiten añadir cualquier dato dándole formato y características. Pese a que su uso es un poco más complicado al habitual, es totalmente recomendable.

```
<ListView
        android:id="@android:id/list"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />
<TextView
        android:id="@+id/android:empty"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:text="@string/not_contacts_yet"/>
```

Utilizando las etiquetas correctamente, conseguimos que el sistema operativo muestre el *TextView* con id “andoid:empty” cuando la *ListView* no tiene objetos.

Utilizar una *ListView* implica que nuestra actividad deberá descender de *ListActivity*. Para cada tipo de *ListView* deberemos de crear un adaptador. En este caso el adaptador es para los contactos.

```
private class ContactsAdapter extends BaseAdapter {
        private final Activity actividad;
        private final List<Contact> lista;
        public ContactsAdapter(Activity activity, List<Contact> lista) {
                super();
                this.actividad = activity;
                this.lista = lista;
```

```
        }
        @Override
        public int getCount() {
                return lista.size();
        }
        @Override
        public Object getItem(int position) {
                return lista.get(position);
        }
        @Override
        public long getItemId(int position) {
                return position;
        }
        @Override
        public View getView(int position, View convertView, ViewGroup parent) {
                LayoutInflater inflater = actividad.getLayoutInflater();
                View view = inflater.inflate(R.layout.contact_icon, null, true);
                view.setTag(lista.get(position));
                view.setLongClickable(true);
                TextView name = (TextView) view.findViewById(R.id.name);
                name.setText(lista.get(position).getAlias());
                return view;
        }
}
```

Cuando creamos el *BaseAdapter*, este trata uno a uno todos los objetos que se le han pasado añadiendo los datos relevantes a una vista. En este caso, para tener toda la información accesible en la vista, hemos añadido el objeto al que representa como una *Tag*.

Finalmente cambiamos el texto de la vista para que tenga el mismo nombre que el contacto al que representa.

4.1.4.3 Capa de persistencia

La gran mayoría de la capa de persistencia es accedida mediante un proceso en background. Eso se hace para que el proceso que controla la interfaz del usuario no se quede bloqueado ni de la sensación de ir lento.

4.1.4.3.1 SharedPreferences

Para facilitar el uso de SharedPreferences se ha creado una clase la cual se detalla a continuación:

```
public class SharedPrefHelper {

        private SharedPreferences sp;

        private static SharedPrefHelper instance = null;

        private SharedPrefHelper(Context context) {

                this.sp = context.getSharedPreferences("LETTER",
                Context.MODE_MULTI_PROCESS);

        }

        public static SharedPrefHelper getInstance(Context context) {

                if (instance == null) instance = new SharedPrefHelper(context);

                return instance;

        }

        public long getChatCount() {

                long res = sp.getLong("CHAT_COUNT", 0);

                sp.edit().putLong("CHAT_COUNT", res + 1).commit();

        return res;

        }

        public long getMessageCount() {

                long res = sp.getLong("MESSAGE_COUNT", 0);

                sp.edit().putLong("MESSAGE_COUNT", res + 1).commit();

                return res;

        }

        public String getUserId() {

                return sp.getString("USER_ACCOUNT", null);

        }

        public void setUserId(String val) {

                sp.edit().putString("USER_ACCOUNT", val).commit();

        }

        public String getUserAlias() {
```

```
            return sp.getString("USER_ALIAS", null);
        }
        public void setUserAlias(String val) {
            sp.edit().putString("USER_ALIAS", val).commit();
        }
}
```

4.1.4.3.2 SQLite

SQLite es un sistema de gestión de bases de datos relacional compatible con transacciones atómicas, consistentes, aisladas y duraderas, lo que significa que es compatible con ACID. Además es peculiarmente ligero, por lo que es perfecto para nuestra aplicación en el cliente.

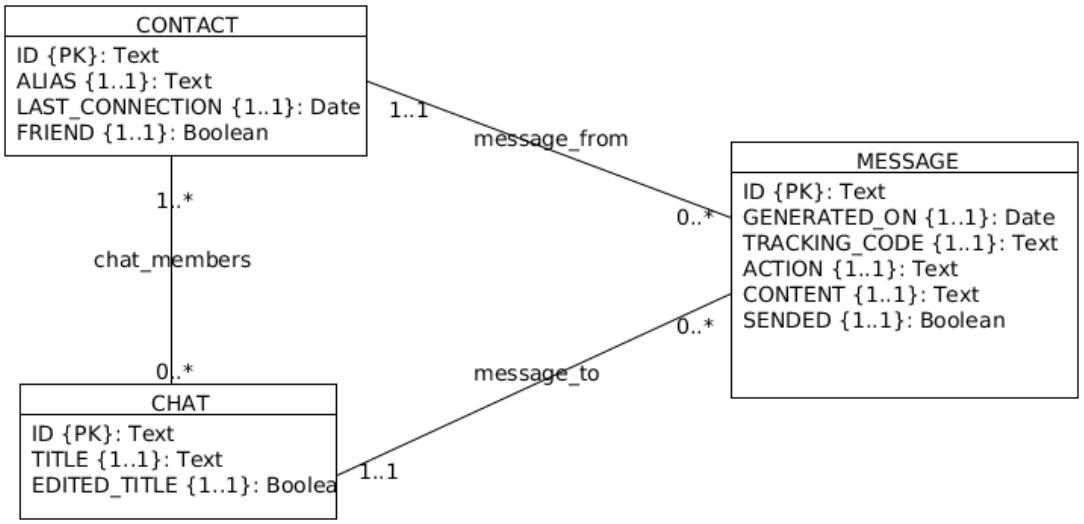

Ilustración 4-19 – Base de datos del cliente

4.1.4.3.3 Conexión con el servidor

Todas las conexiones con el servidor se realizan mediante esta clase. Esta clase es la que se encarga de controlar desde la dirección del servidor, hasta los protocolos utilizados.

En este caso enseñamos la función *getContact*, con la cual recuperamos la información de un contacto del servidor.

```
public Contact getContact(String contactId) throws NoExistentUserException {
        Contact res = null;
        DatagramSocket socket = null;
        try {
            boolean sended = false;
            Message m = new Message("serverlet", userId, "serverlet",
            Controller.getDate(), null, Message.actions.get_contact,
            "<id>" + contactId + "</id>");
```

```
            DatagramPacket request = null;
            byte[] messagebytes;
            while (!sended) {
                socket = new DatagramSocket();
                socket.setSoTimeout(10000);
                messagebytes = m.toXml().getBytes();
                request = new DatagramPacket(messagebytes,
                messagebytes.length, InetAddress.getByName(SERVERURL),
                SERVERPORT);
                socket.send(request);
                Log.d(context.getString(R.string.app_name), "SERVER:
                        getContact enviado: " + m.toXml());
                messagebytes = new byte[MAXMESSAJELENGHT];
                request = new DatagramPacket(messagebytes,
                messagebytes.length);
                try {
                        socket.receive(request);
                } catch (SocketTimeoutException e) {
                        continue;
                }
                sended = true;
            }
            //Analizamos los datos de la respuesta
            Response resp = parseMessage(request);
            if (resp.content.equals("no_match"))
                    throw new NoExistentUserException();
            res = new Contact(contactId, resp.lastConnection);
            res.setAlias(resp.alias);
        } catch (BadAliasValueException | BadAccountValueException e) {
            Log.e(context.getString(R.string.app_name), "SERVER: Problems
            recovering contact", e);
        } catch (BadToValueException | BadFromValueException | BadIdValueException
        | BadDateValueException e) {
            Log.e(context.getString(R.string.app_name), "SERVER: Error creando
            mensaje para addContact", e);
```

```
        } catch (UnknownHostException e) {
                Log.e(context.getString(R.string.app_name), "SERVER: No se ha
        encontrado el servidor", e);
        } catch (IOException e) {
                Log.e(context.getString(R.string.app_name), "SERVER: excepción de
                imput/output", e);
        } finally {
                if (socket != null) socket.close();
        }
        return res;
}
```

4.2 ARQUITECTURA DEL SERVIDOR

Igual que en el cliente, para la aplicación servidor se utilizara la programación por capas.

4.2.1 Nivel de presentación

En este caso el servidor no tiene interfaces visuales. Las interfaces del servidor servirán para que nuestros clientes puedan mandarte peticiones. Estas interfaces sirven para que el cliente no depende de la implementación interna del servidor. De esta forma, el servidor podría cambiar el servicio de base de datos sin que ningún cliente percibiera los cambios.

4.2.2 Nivel de aplicación

El nivel de aplicación ofrece a las aplicaciones la posibilidad de acceder a los servicios de las demás capas y define los protocolos que utilizan las aplicaciones.

Se reciben los mensajes y se analizan para obtener las peticiones. Una vez la petición es identificada se procesa. Por lo general, este procesamiento implica al nivel de persistencia. Una vez realizada toda la computación se envía un mensaje al cliente para notificar el resultado de la petición.

4.2.3 Nivel de persistencia

El servidor se encarga de guardar los siguientes datos:

Usuario

- Id
- Contraseña
- Alias
- Última conexión

Chat

- Id
- Titulo

Mensaje

- Id
- From
- To
- Código de rastreo
- Fecha de generación
- Acción
- Contenido

4.2.4 Implementación detallada

4.2.4.1 Capa de presentación

Gracias a los comportamientos predefinidos podemos programar el servidor teniendo en cuenta únicamente la funcionalidad de éste, sin tener que preocuparnos por tener que tratar los errores.

```
-module(main).

-behaviour(gen_server).

-define(SERVER, ?MODULE).

-record(state, {socket}).

start() -> gen_server:start({local, ?SERVER}, ?MODULE, [], []).

stop() -> gen_server:cast(?SERVER, stop).

init(_Args)->

        db_interface:install(),

        db_interface:start(),

        {ok,Socket} = gen_udp:open(?LOCALPORT),

        {ok, #state{socket = Socket}}.

handle_info(Info, State)->

        {udp, Socket, Ip, PortNumber, Packet} = Info,

        spawn(req_handler, peticion, [udp, Socket, Ip, PortNumber, Packet]),

        {noreply, State}.
```

Como podemos comprobar, en este caso se sigue el comportamiento “gen_server”, el cual crea un servidor que atenderá peticiones.

Las instrucciones *start* y *stop* son utilizadas para poner en marcha y parar el servicio y son las únicas instrucciones que deberían de ser llamadas directamente por el usuario.

Las instrucciones como *init* y *handle_info* son instrucciones que llama el comportamiento de *gen_server* cuando un evento sucede. Por ejemplo, cuando llamamos a la función *start*, internamente, además de hacer otras tareas, se está llamando a la instrucción *init*. Como comentábamos en la introducción, cada nodo de Erlang posee una bandeja de entrada de mensajes donde cada mensaje se compara con los patrones de mensajes que se esperan. En este caso, cuando un mensaje recibido no encaja en ningún patrón, dicho mensaje es enviado a la función *handle_info*. Desde esta función es desde donde recibimos los paquetes transmitidos por el cliente y lo podemos tratar en nuestro servidor.

4.2.4.2 Capa de aplicación

4.2.4.2.1 Req_handler

```
case Action of
        "sign_up"       →       % Código para sign_up %;
        "sign_in"       →       % Código para sign_in %;
        …               →       ...
end,
ok = gen_udp:send(Socket, Ip, PortNumber, Message).
```

Como hemos nombrado antes, Erlang fue diseñado para plataformas de telecomunicaciones en las que había mucha concurrencia, pero cada proceso realizaba una cantidad muy pequeña de computación. En nuestro caso sucede lo mismo, el sistema está preparado para un alto nivel de concurrencia pero cada petición tiene una cantidad muy pequeña de computación.

Dependiendo del caso se hace una búsqueda en la base de datos, una modificación o una eliminación. Los mensajes de respuesta son rellenados dependiendo del resultado de la operación en la base de datos. Finalmente todas las operaciones envían un mensaje al cliente.

4.2.4.2.2 Descomponiendo los mensajes

Erlang ofrece una librería para descomponer el texto en formato *xml*, *xmerl*. *Xmerl* escanea los elementos y los devuelve en un formato tratable en el lenguaje Erlang.

El lenguaje de etiquetado *xml* puede ser muy extenso y tener muchas propiedades, es por esto que *xmerl* devuelve mucha información, mucha más de la que necesitamos. Como nuestros mensajes siempre tienen los mismos campos hay mucha información que podemos descartar.

Uncompose_element se encarga de eliminar los campos devueltos por *xmerl* que no nos hacen falta y ordena los campos requeridos para tener una forma más cómoda de trabajar con ellos.

```
uncompose_element([]) -> [];
uncompose_element({xmlElement, Etiqueta, Etiqueta, _CV1,  _XmlNamespace,  _Parents, _NumElemento, Atributes, Elementos,  _CV2, _Ruta, _Desconocido}) ->
        {Etiqueta, atributes, uncompose_element(Atributes), elements,
        uncompose_element(Elementos)};
uncompose_element([H|T]) ->
        [uncompose_element(H) | uncompose_element(T)];
uncompose_element({xmlText, _Parents, _NumElemento, _CV, Texto,text}) ->
        {text,Texto};
uncompose_element({xmlAttribute, Atribute, _CV1, _CV2, _CV3, _Parents, _NumElemento, _CV4, Value, _Boolean}) ->
        {Atribute,Value};
```

```
uncompose_element(Content) →
        Content.
```

Pese a que la información ya está acotada, todavía se sigue tratando de forma genérica. Es la función *parse_message* la que se encarga de dar forma a la información que estamos tratando para que tenga el resultado esperado; un mensaje con todos sus campos bien definidos.

```
parse_message([{message, atributes, Atributes, elements, Elements} | _Rest])->
        {atributes, to, To, from, From, id, Id, 'generated.on', Date, 'tracking.code', TCode, action, Action} =parse_atributes(Atributes),
        Content = parse_elements(Elements),
        {message, to, To, from, From, id, Id, 'generated.on', Date, 'tracking.code', TCode, action, Action, Content}.
parse_atributes([ _XmlVersion, {to, To}, {from, From}, {id, Id}, {'generated.on', Date}, {'tracking.code', Tcode}, {action, Action}]) ->
        {atributes, to, To, from, From, id, Id, 'generated.on', Date, 'tracking.code', TCode, action, Action}.
parse_elements([])->[];
parse_elements([{text, Content}|Rest])->
        [{text, Content} | parse_elements(Rest)];
parse_elements([{Etiqueta, atributes, _atributos, elements, [{text, Valor}]}|Rest])->
        [{Etiqueta, Valor}| parse_elements(Rest)].
```

Después de todo el tratamiento de la información, este es el resultado:

```
{message, to, To, from, From, id, Id, 'generated.on', Date, 'tracking.code', TCode, action, Action, Content}
```

4.2.4.3 Capa de persistencia

Mnesia fue desarrollada por *Ericsson* para sistemas distribuidos de tiempo real con alta disponibilidad. No fue desarrollada para sustituir los sistemas *SQL*, sino que fue implementada para dar soporte a Erlang donde la persistencia es necesaria.

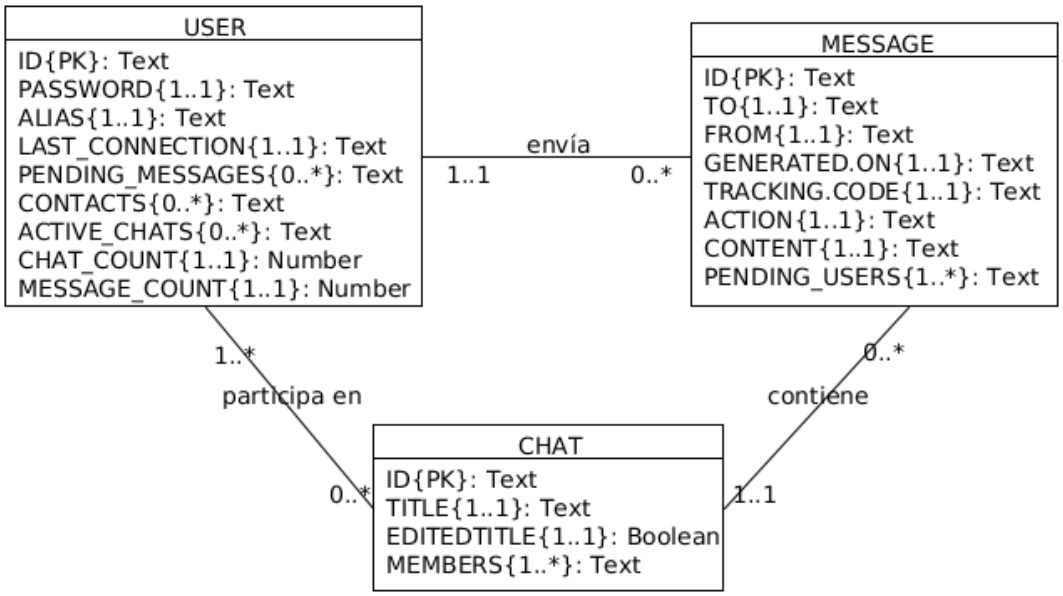

Ilustración 4-20 – Base de datos del servidor

```
record(user, {id, password, alias, last_connection, pending_messages, contacts, active_chats, chat_count, message_count}).

record(chat, {id, title, editedTitle, members}).

record(message, {id, to, from, 'generated.on', 'tracking.code', action, content, pending_users}).
```

5 CONCLUSIONES

En este trabajo se ha tratado de abordar el problema de implementar una aplicación en la que se comuniquen lenguajes de diferentes paradigmas. En nuestro caso se ha utilizado Erlang en el ejemplo de una aplicación de mensajería instantánea.

Se ha creado una aplicación de mensajería que cumplía con los requisitos especificados en la introducción de este documento. Dicha aplicación está únicamente disponible para Android, pero gracias a las decisiones tomadas en la comunicación cliente – servidor se podrán desarrollar nuevos clientes en distintas plataformas sin tener que hacer modificaciones significativas en el servidor.

Se han utilizado *AndroidStudio* y Eclipse como herramientas principales de desarrollo, utilizando las distintas librerías disponibles que nos han facilitado el trabajo de una forma notoria.

5.1 POSIBLES AMPLIACIONES

En una futura mejora de la aplicación, se podrían dar las siguientes ampliaciones:

- Poder añadir a más de un contacto a un chat.
- Poder enviar imágenes, archivos y vídeos en un chat.
- Añadir imágenes a los perfiles de los usuarios.
- Crear un cliente web.
- Cifrar las comunicaciones cliente – servidor.

6 BIBLIOGRAFÍA

[1] Tomás Gironés, Jesús (2015). El gran libro de Android. Barcelona: marcombo.

[2] Cesarini, Francesco y Thompson, Simon (2009). Erlang Programming. Estados Unidos de América: O'Reilly Media, inc.

[3] Armstrong, Joe (2007). A history of Erlang. HOPL III, 6 (1), 6-26.

[4] Google Inc. Google Material Design. https://material.google.com/.

[5] Ericsson AB (1999). Erlang/OTP 18. http://erlang.org/erldoc.

[6] Google Inc (2016). Android API. https://developer.android.com/reference/packages.html.

[7] WebUsable.com. El significado de los colores. http://www.webusable.com/coloursMean.htm.

[8] WebsA100 (2015). Guía del color en marqueting y branding. http://www.websa100.com/bloq/estrategias-de-marketing-online-guia- de-color-para-marketing.

7 ANEXOS

7.1 ANEXO I: CASOS DE USO

7.1.1 Identificación

Actores	Usuario no identificado.
Propósito	El usuario debe identificarse en el sistema.
Resumen	El usuario debe identificarse en el sistema para poder utilizar todas las características.
Pre condiciones	-
Post condiciones	La cuenta debe almacenarse en remoto y el usuario debe de estar identificado en local.
Incluye	-
Extiende	-

7.1.2 Crear cuenta

Actores	Usuario no identificado.
Propósito	El usuario puede crear una cuenta en el sistema.
Resumen	El usuario crea una cuenta con un nombre único que todavía no está en uso.
Pre condiciones	-
Post condiciones	La cuenta se almacena en el sistema tanto en local como en remoto.
Incluye	-
Extiende	Identificación.

7.1.3 Cargar cuenta

Actores	Usuario no identificado.
Propósito	El usuario puede acceder a su cuenta.
Resumen	El usuario puede acceder a una cuenta ya creada con anterioridad mediante su nombre de usuario y su contraseña.
Pre condiciones	El usuario ya posee una cuenta.
Post condiciones	Se autentica correctamente en el sistema. Los datos de la cuenta son cargados en local.
Incluye	-
Extiende	Identificación.

7.1.4 Añadir contacto

Actores	Usuario identificado.
Propósito	El usuario puede añadir contactos.
Resumen	El usuario puede añadir contactos conociendo su identificador en el sistema.
Pre condiciones	El contacto tiene que estar en el sistema.
Post condiciones	La información del contacto se guarda en local.
Incluye	Identificación.
Extiende	-

7.1.5 Modificar contacto

Actores	Usuario identificado.
Propósito	El usuario puede modificar un contacto.
Resumen	Se puede modificar el alias de un contacto.
Pre condiciones	El contacto tiene que estar añadido como amigo en local.
Post condiciones	Las modificaciones se guardan en local.
Incluye	Identificación.
Extiende	-

7.1.6 Eliminar contacto

Actores	Usuario identificado.
Propósito	El usuario puede eliminar un contacto de su lista de contactos.
Resumen	El usuario puede eliminar un contacto de su lista de contactos.
Pre condiciones	El contacto tiene que estar en la lista local de contactos, por lo que antes tiene que haber sido añadido.
Post condiciones	Las modificaciones se guardan en local.
Incluye	Identificación.
Extiende	-

7.1.7 Añadir chat

Actores	Usuario identificado.
Propósito	El usuario puede crear un chat.
Resumen	El usuario puede crear un chat con un contacto de su lista de contactos.
Pre condiciones	El chat se crea con un contacto de la lista local de contactos.
Post condiciones	El chat se guarda en local y en el servidor.
Incluye	Identificación.
Extiende	-

7.1.8 Modificar chat

Actores	Usuario identificado.
Propósito	El usuario puede modificar un chat.
Resumen	El usuario puede modificar los datos de un chat.
Pre condiciones	El chat tiene que estar en local.
Post condiciones	Las modificaciones del chat se guardan en local.
Incluye	Identificación.
Extiende	-

7.1.9 Eliminar chat

Actores	Usuario identificado.
Propósito	El usuario puede eliminar un chat.
Resumen	El usuario puede eliminar un chat de su lista de chats activos.
Pre condiciones	El chat tiene que estar en local.
Post condiciones	Las modificaciones se realizan en local y en el servidor.
Incluye	Identificación.
Extiende	-

7.1.10 Enviar mensaje

Actores	Usuario identificado.
Propósito	El usuario puede enviar un mensaje.
Resumen	El usuario puede enviar un mensaje a otro usuario mediante un chat.
Pre condiciones	El usuario tiene que tener un chat creado para poder enviar el mensaje.
Post condiciones	El mensaje se guarda en local y en el servidor.
Incluye	Identificación.
Extiende	-

7.2 Anexo II: Índice de ilustraciones