



**UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA**

GRADO EN INGENIERÍA INFORMÁTICA

TECNOLOGÍA ESPECÍFICA DE INGENIERÍA DEL SOFTWARE

TRABAJO FIN DE GRADO

Aplicación de Mensajería instantánea Android

Antonio Laguna Rivera

Septiembre, 2014



UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

DEPARTAMENTO DE TECNOLOGÍAS Y SISTEMAS DE
INFORMACIÓN

TECNOLOGÍA ESPECÍFICA DE INGENIERÍA DEL SOFTWARE

TRABAJO FIN DE GRADO

Aplicación de Mensajería instantánea Android

Autor: Antonio Laguna Rivera
Director: Dr. Macario Polo Usaola

Septiembre, 2014

TRIBUNAL:

Presidente:

Vocal:

Secretario:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE

VOCAL

SECRETARIO

Fdo.:

Fdo.:

Fdo.:

RESUMEN

Desde la aparición de internet, uno de las grandes mejoras en la calidad de vida de las personas ha sido la posibilidad de comunicarse con cualquier persona a través de internet, fuese cual fuese su situación física.

Con la aparición de los *Smartphones* esta comunicación ha mejorado bastante, permitiendo a cualquier usuario de uno de estos terminales acceder a internet y comunicarse de forma directa con cualquier usuario. De hecho, las aplicaciones más exitosas en el mercado de la tecnología móvil son las de mensajería instantánea.

Actualmente existen varias alternativas para llevar a cabo una comunicación entre distintos usuarios, aunque muchas de ellas han sido criticadas por la falta de privacidad de los datos del usuario, ofreciendo a los demás usuarios información no deseada como la hora de la última conexión y el hecho de recibir los mensajes que se han enviado.

En el presente Trabajo Fin de Grado se muestra el desarrollo de TalkMe, una aplicación de mensajería instantánea para Android, que hace especial hincapié en ocultar los datos y marcas de conexión de los usuarios frente a los demás usuarios del sistema.

Para ello se ha tomado como idea base los actuales sistemas de mensajería instantánea, eliminando ciertos elementos que el usuario podría interpretar como intrusismo a su intimidad.

El objetivo es la creación de una aplicación completa de mensajería instantánea comparable a las existentes en cuanto a funcionalidad, con cierto matiz de privacidad de datos que la diferenciaría del resto.

ABSTRACT

Since the appearance of internet, one of the great improvements in the quality of life of people has been able to communicate with anyone on the internet independently of their geographical location.

With the appearance of smartphones this communication has greatly improved, allowing any user of one of these phones to access the internet and communicate directly with any user. In fact, the most successful applications in the mobile technology market is instant messaging.

Currently there are several alternatives for carrying out communication between different users, although many have been criticized for their privacy policy user data, providing users other unwanted information such as the time of last connection and the fact receive messages that have been sent.

In this final Project will be shown the development of "TalkMe", an Instant Messaging (IM) application for Android, which places special emphasis on data hiding and brands Connection users to other users of the system.

For this, it has been based in current IM systems removing items that the user could be interpreted as intrusion in his privacy.

The goal is to create a complete instant messaging application comparable to existing applications in terms of functionality, with a tinge of data privacy, differentiating it from the rest.

DEDICATORIA

A mis padres Manuel Antonio y Mercedes, gracias a los cuales soy quien soy, y sin los cuales no habría podido llevar a cabo el presente TFG.

Agradecimientos al director del presente TFG, Macario Polo, por su paciencia y dedicación.

Agradecimiento a mis compañeros por estar presentes en este camino, y en especial a Juan José, Víctor y Carlos por su ayuda en el momento más difícil.

Índice

Índice de Figuras	I
1. Introducción.....	1
1.1. Situación	1
1.2. Motivación	3
1.3. Alcance	4
1.4. Objetivos	4
1.4.1. Objetivos específicos	6
1.5. Productos Similares	7
2. Tecnología	11
2.1. Servidor.....	11
2.1.1. Características	11
2.1.2. Sintaxis.....	14
2.1.3. Rendimiento.....	20
2.1.4. Resumen y conclusiones.....	22
2.2. Cliente	23
2.2.1. Características	24
2.2.2. Arquitectura	27
2.2.3. Compilación.....	31
3. Análisis del Sistema	33
3.1. Casos de Uso.....	34
3.1.1. Casos de Uso del Cliente	35
3.1.2. Casos de uso del Servidor	41
3.1.3. Diagrama de casos de uso	45
3.2. Análisis de requisitos	46
3.2.1. Requisitos Funcionales	46
3.2.1.1. Requisitos Funcionales Servidor	46
3.2.1.2. Requisitos Funcionales Cliente.....	47
3.2.2. Requisitos no funcionales	47
3.2.2.1. Requisitos No Funcionales Servidor.....	48
3.2.2.2. Requisitos No Funcionales Cliente	48
4. Desarrollo y etapas	49

4.1. Riegos iniciales y cuestiones preliminares	50
4.2. Ciclos	51
4.2.1. Ciclo 1: Prueba de Concepto. Envío de mensajes	51
Determinar Objetivos	51
Análisis de riesgos.....	52
Desarrollar, Verificar y Validar	52
Planificación.....	56
4.2.2. Ciclo 2: Instalación y registro. Persistencia en el Servidor	57
Determinar Objetivos.....	57
Análisis de riesgos.....	57
Desarrollar, Verificar y Validar	58
Planificación:.....	72
4.2.3. Ciclo 3: Mejora en el envío de mensajes.	74
Determinar Objetivos	74
Análisis de riesgos.....	74
Desarrollar, Verificar y Validar:	75
Planificación:.....	79
4.2.4. Ciclo 4: Selección de contactos. Interfaz principal.....	81
Determinar Objetivos	81
Análisis de riesgos.....	81
Desarrollar, Verificar y Validar:	82
Planificación.....	90
4.2.5. Ciclo 5: Recepción de mensajes	91
Determinar Objetivos	91
Análisis de riesgos.....	91
Desarrollar, Verificar y Validar:	92
Planificación:.....	97
4.2.6. Ciclo 6: Persistencia en el cliente	98
Determinar Objetivos	98
Análisis de riesgos.....	98
Desarrollar, Verificar y Validar:	98
Planificación:.....	104
4.2.7. Ciclo 7: Mejora en la interfaz de conversación	105

Determinar Objetivos	105
Análisis de riesgos.....	106
Desarrollar, Verificar y Validar:	106
Planificación.....	111
4.2.8.Ciclo 8: Ciclo de vida del cliente y Action Bar	112
Determinar Objetivos	112
Análisis de riesgos.....	113
Desarrollar, Verificar y Validar:	113
Planificación.....	120
4.2.9.Ciclo 9: Cambio en el sistema de recepción de mensajes.....	122
Determinar Objetivos	122
Análisis de riesgos.....	124
Planificación:.....	130
4.2.10.Ciclo 10: Conversaciones en Grupo	131
Determinar Objetivos	131
Análisis de riesgos.....	131
Desarrollar, Verificar y Validar:	132
Planificación.....	144
5. Banco de pruebas.....	147
6. Conclusiones.....	151
6.1. Objetivos funcionales	151
6.2. Objetivos didácticos.....	153
6.3. Trabajos futuros	154
7. Referencias	157
8. ANEXOS	161
8.1. ANEXO 1: Manual de usuario.....	161
Registro de nuevo usuario	161
Loguearse en la aplicación	163
Iniciar una conversación.....	165
Crear un grupo de conversación.....	168
8.2. ANEXO 2: Instalación del servidor.....	173

Índice de Figuras

Figura 1: Cuota de mercado 3Q (IDC 2013)	2
Figura 2: Actividades realizadas en Internet móvil (IAB Spain)	2
Figura 3: Diseño del proyecto	5
Figura 4: Vista general de conexiones del proyecto.....	5
Figura 5: Características WhatsApp	7
Figura 6: Características Line	8
Figura 7: Características Viber	9
Figura 8: Características Telegram.....	10
Figura 9: Ejemplo JSP	12
Figura 10: Proceso de generación de página JSP	13
Figura 11: HTML generado con JSP	13
Figura 12: Rendimiento JSP	20
Figura 13: Puestos de trabajo JSP.....	21
Figura 14: JSP vs PHP.....	21
Figura 15: Ventas de smartphones 3Q 2013. Fuente: <i>IDC 2013[2]</i>	23
Figura 16: Estructura de directorios de un proyecto Android	26
Figura 17: Arquitectura Android	27
Figura 18: Proceso de compilación Android.	
http://developer.android.com/tools/building/index.html	31
Figura 19: Proceso de compilación Android (2)	
http://developer.android.com/tools/building/index.html	32
Figura 20: Ciclo de vida clásico	33
Figura 21: Diagrama de casos de uso del cliente.....	45
Figura 22: Ejemplo de desarrollo en espiral	50
Figura 23: Objetivo ciclo 1	51
Figura 24: Diagrama de clases del Cliente (ciclo 1).....	52
Figura 25: Diagrama de clases del Servidor (ciclo 1)	53
Figura 26: Interfaz Cliente (ciclo 1)	54
Figura 27: Enviar Mensaje. GUIConversacion.java.....	54
Figura 28: codificar. GestorMensaje.java.....	55
Figura 29: enviar. Envio.java	55
Figura 30: recibir.jsp	56
Figura 31: Diagrama de clases Cliente. Ciclo 2	59
Figura 32: Diagrama de clases Servidor. Ciclo 2	60
Figura 33: Experiencia de usuario. Ciclo 2	61
Figura 34: Reproducción de la Figura 30	61
Figura 35: BBDD Servidor. Ciclo 2	62
Figura 36: Conectar. Agente.java	62
Figura 37: Reproducción de la Figura 31	63
Figura 38: Login.jsp	64
Figura 39: guardar. GestorUsuario.java	64
Figura 40: insertarEnBaseDeDatos. UsuarioBBDD.java	64

Figura 41: registrar.jsp.....	65
Figura 42: existeEnBaseDeDatos. UsuarioBBDD.java.....	66
Figura 43: loggin. GestorUsuario.java	66
Figura 44: Patrón Singleton. Agente.java.....	66
Figura 45: Comportamiento Cliente. Ciclo 2	67
Figura 46: comprobar instalación. GUIBienvenida.java	68
Figura 47: comprobarInstalacion. Instalacion.java.....	68
Figura 48: login. GUILogin.java	69
Figura 49: AsyncTask. UserLoginTask.java.....	70
Figura 50: envio de credenciales. GestorCredenciales.java	71
Figura 51: Diagrama de clases UML cliente	73
Figura 52: Diagrama de clases UML servidor.....	73
Figura 53: calcularCRC32. GestorMensajes.java.....	74
Figura 54: Máquina de estados para la comprobación de envío. Concepto	75
Figura 55: enviar. GestorMensaje.java.....	76
Figura 56: gestionar respuesta. EnviarMensajeHilo.java	76
Figura 57: recibir mensajes en el Servidor y responder con el Hash. recibir.jsp	76
Figura 58: limpiarRespuesta. Envio.java.....	77
Figura 59: BBDD Servidor. Ciclo 3	77
Figura 60: Diagrama de clases Servidor. Ciclo 3	78
Figura 61: Carga de Contactos. Contactos.java.....	82
Figura 62: Selección de Contacto.....	83
Figura 63: Adaptador de Contactos. ItemAdapter.java	83
Figura 64: lanzar Interfaz de selección de contacto. GUIPanelPrincipal.java	84
Figura 65: Recoger resultado de activity. GUIPanelPrincipal.java.....	84
Figura 66: parcelización de una clase. Contacto.java.....	86
Figura 67: Interfaz principal. Ciclo 4	87
Figura 68: abrir conversación. GUIPanelPrincipal.java	88
Figura 69: creación y actualización de conversación. GUIConversación.java	89
Figura 70: ReceptorMensajes.java	92
Figura 71: Envío de petición de recepción de mensajes. GestorMensaje.java.....	93
Figura 72: Runnable de actualización	94
Figura 73: creación de Handle.....	94
Figura 74: Creación del actualizador	95
Figura 75: actualizar. Actualizador.java.....	95
Figura 76: actualización de GUIConversacion. GUIConversacion.java	96
Figura 77: Control de conversación abierta. GUIPanelPrincipal.java.....	96
Figura 78: Filtro para la recepción de nuevos Intents en GUIConversación.	
Manifest.xml.....	96
Figura 79: BBDD Cliente (Ciclo 6).....	99
Figura 80: Agente de bases de datos del cliente. AgenteBBDD.java.....	100
Figura 81: Adaptador de mensajes a base de datos. MensajeBBDD.java.....	101
Figura 82: de cursor a mensaje. MensajeBBDD.java.....	102
Figura 83: getMensajes modificado. GestorMensaje.java.....	102

Figura 84: enviar modificado. GestorMensaje.java.....	102
Figura 85: cargar conversaciones almacenadas en la base de datos.	
CargadorConversaciones.java	103
Figura 86: Envío de mensajes pendientes. ReceptorMensajes.java	103
Figura 87: Ejemplo conversación WhatsApp	105
Figura 88: Ejemplo conversación Line	106
Figura 89: activity_conversacion.xml	107
Figura 90: elemento_conversacion2.xml.....	108
Figura 91: converAdapter.java	109
Figura 92: Ejemplo de conversación con la interfaz mejorada	110
Figura 93: Ciclo de vida de Actividad Android. Fuente: http://developer.android.com/training/basics/activity-lifecycle/starting.html	112
Figura 94: Modificación del comportamiento de los botones de salida.	
GUIPanelPrincipal.java	113
Figura 95: Primitiva al sistema para cerrar la aplicación	114
Figura 96: Comprobación de estado de aplicación para notificar al usuario.....	115
Figura 97: lanzarNotificacion. GUIPanelPrincipal.java.....	116
Figura 98: Eliminación de notificaciones de usuario	116
Figura 99: Implementación de ActionBarActivity	117
Figura 100: menu/guipanelprincipal.xml	118
Figura 101: Selección de opción del menú.....	119
Figura 102: Estado GUIPanelPrincipal. Ciclo 8.....	119
Figura 103: Proceso de envío de mensajes por GCM	123
Figura 104: Comprobación de credenciales GCM. GUIPanelprincipal.java	125
Figura 105: TareaRegistroGCM. Registro en servidor GCM	125
Figura 106: registroServidor. GUIPanelPrincial.java.....	126
Figura 107: GCMBroadcastReceiver	126
Figura 108: GCMIntentService	127
Figura 109: recibir.jsp. Ciclo 9	128
Figura 110: GCMBroadcast. Servidor	129
Figura 111: UML Grupo	132
Figura 112:GUI_Nuevo_Grupo. Interfaz	133
Figura 113: enviarPeticionCreacionGrupo. GUI_Nuevo_Grupo.java	133
Figura 114: crearGrupo. Grupo.java.....	134
Figura 115: enviar. CrearGrupo.java	134
Figura 116: Base de datos del Servidor. Ciclo 10	135
Figura 117: crearGrupo.jsp. Recepción de petición de creación de grupo.....	136
Figura 118: enviarNotificacion. Grupo.java (Servidor)	136
Figura 119: Confirmar la creación del grupo al creador. GCMConfirmacionGrupo.java	137
Figura 120: Filtro de mensajes en el cliente. GCMIntentService.java (Cliente).....	137
Figura 121: almacenarGrupo. GCMIntentService.java	138
Figura 122: Crear un nuevo grupo ya confirmado con la función actualizar. Actualizador.java	138

Figura 123: Diagrama Entidad-Relación Cliente. Implementación de grupos.....	138
Figura 124: GUIGrupo.java. Interfaz gráfica	139
Figura 125: Recuperación de mensaje de grupo en el servidor. recibirMensajeGrupo.jsp	140
Figura 126: recuperación de participantes en un grupo.....	141
Figura 127: Envío de mensaje de grupo del servidor al cliente. GCMMensajeGrupo.java	141
Figura 128: filtro de mensajes entrantes del servidor GCM.....	142
Figura 129: Recuperación de mensaje de grupo en el cliente. Función leerMensajeGrupo. GCMIntentService.java	143
Figura 130: Almacenado de mensaje de grupo y actualización de la GUI.....	143
Figura 131: Diagrama de clases UML del Cliente. Ciclo 10	145
Figura 132: Diagrama de clases UML del servidor. Ciclo 10	146
Figura 133: TestLogin	148
Figura 134: TestRegistro	148
Figura 135: TestEnvioMensajes	149
Figura 136: Resultado Banco de pruebas	149
Figura 137: Pantalla de bienvenida de la aplicación	161
Figura 138: Interfaz de registro de la aplicación	162
Figura 139: Interfaz principal de TalkMe.....	162
Figura 140: Pantalla de bienvenida. Opción Conectarse	163
Figura 141: Formulario de login.....	164
Figura 142: Botón de nueva conversación	165
Figura 143: Selección de contacto para iniciar conversación.....	166
Figura 144: Panel principal con conversación vacía	166
Figura 145: Conversación vacía	167
Figura 146: Mensaje enviado correctamente	167
Figura 147: Botón Nuevo Grupo	168
Figura 148: Creación de nuevo grupo	169
Figura 149: Grupo creado.....	169
Figura 150: Interfaz de conversación en grupo vacía.....	170
Figura 151: Interfaz de conversación en grupo con el primer mensaje del creador	171
Figura 152: Modificación de la ruta de la base de datos del servidor	173
Figura 153: fichero rutas. Modificación de IP	173

1. Introducción

El presente trabajo fin de grado (TFG) desarrolla una aplicación cliente-servidor de mensajería instantánea para dispositivos móviles, basada en el sistema operativo Android.

A continuación se presenta la memoria y documentación del TFG “Cliente de Mensajería instantánea Android”.

En este primer capítulo se detallará la situación actual del sector que abarcamos, las motivaciones para desarrollar este proyecto y sus objetivos.

El segundo capítulo detalla las tecnologías utilizadas para el desarrollo del proyecto, y las razones de su elección.

En el tercer capítulo se analizan las necesidades y requisitos del sistema, base de todo proyecto de ingeniería informática.

En el cuarto capítulo se explica cómo se ha llevado a cabo el desarrollo de la solución, y se detallan aspectos de implementación.

1.1. Situación

Desde la aparición de Internet, la capacidad de comunicar a personas situadas en cualquier punto del planeta, y el intercambio de archivos y otras informaciones a través de Internet ha sido una de las mayores preocupaciones de los desarrolladores de software.

Hasta hace poco, esta funcionalidad estaba limitada al uso de un ordenador y una conexión fija a Internet. Pero con el reciente desarrollo de los smartphones esta capacidad ha sido extrapolada a casi cualquier dispositivo móvil (smartphone, tablet, etc.) con el consiguiente aumento de los usuarios que pueden acceder a estos servicios.

Actualmente los smartphones están muy presentes en la vida cotidiana. Según el informe *Spain Digital Future in Focus/11* de comScore, la penetración de los teléfonos inteligentes en España es del 66% y sólo en 2012, 8 de cada 10 terminales vendidos eran smartphones.

En este nuevo gran grupo de terminales (smartphones) destacan dos sistemas operativos. Por un lado Android (Google), y por otro lado iOS (Apple). Según el informe de *IDC 2013 /21*, el sistema operativo Android ha superado la barrera del 80% de cuota de mercado (Figura 1: Cuota de mercado 3Q (IDC 2013)).

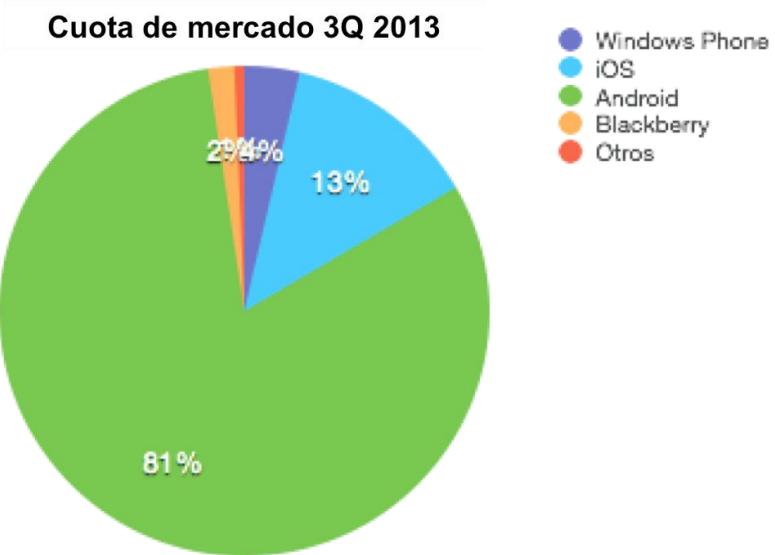


Figura 1: Cuota de mercado 3Q (IDC 2013)

Según el *IV Estudio sobre Mobile Marketing* de IAB Spain y The Cocktail [3] un 91% de los usuarios utiliza su smartphone para chatear, de los cuales un 72'5% lo hace a diario (Figura 2: Actividades realizadas en Internet móvil (IAB Spain)).

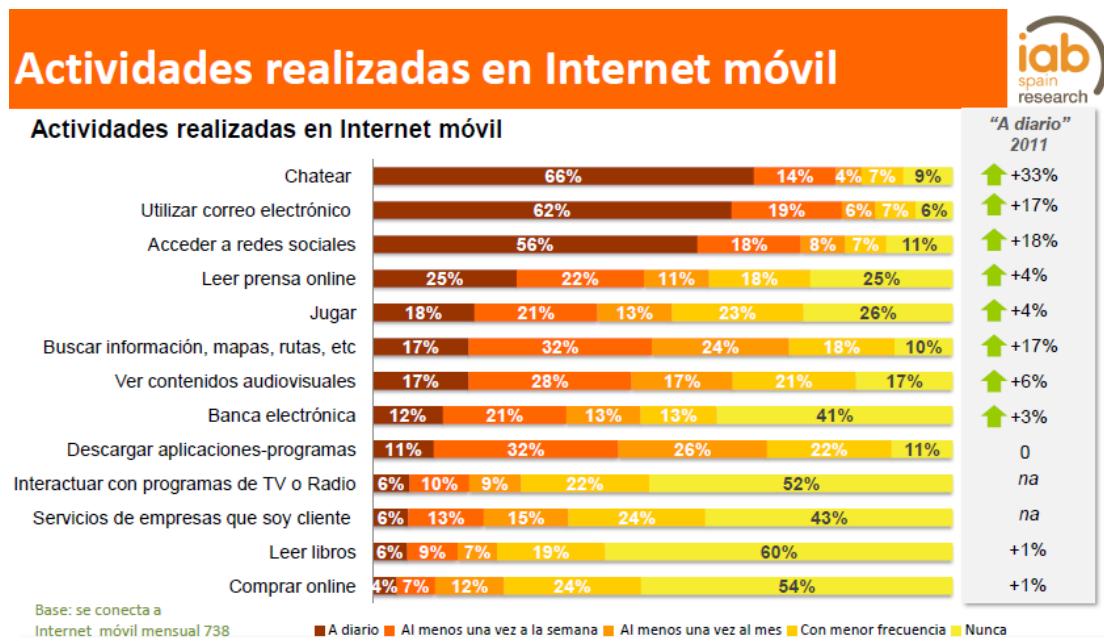


Figura 2: Actividades realizadas en Internet móvil (IAB Spain)

1.2. Motivación

Basándonos en las estadísticas anteriores, y en su proyección de futuro, podemos afirmar que la mensajería instantánea o “chat” es una de las herramientas informáticas más utilizadas actualmente, con millones de usuarios en todo el mundo.

Se pensó en el desarrollo de este Trabajo Fin de Grado (TFG) como una alternativa real a los actuales sistemas de mensajería móvil instantánea, que a pesar de estar muy extendidos entre los usuarios, no destacan en cuanto a la preservación de la intimidad los usuarios. El sistema desarrollado intenta cubrir esta laguna que otros sistemas no han tenido en cuenta. En particular, aplicaciones como Whatsapp y Telegram muestran al resto de usuarios la última hora de conexión, si está bloqueado y si el mensaje ha llegado al destino.

A parte de estas intromisiones en la privacidad de los usuarios, se han producido fallos, muchos de ellos muy mediáticos en la seguridad. Podríamos a continuación destacar unos cuantos de las aplicaciones más conocidas del sector, algunos de los cuales ya han sido subsanados:

-*Hole In WhatsApp For Android Lets Hackers Steal Your Conversations.* [4] – Marzo 2014.

Un agujero en el sistema de seguridad de Whatsapp para Android, permite a los hackers acceder a tus conversaciones guardadas. Whatsapp almacena el historial de conversaciones en la tarjeta SD del teléfono, y esta puede ser accedida por otras aplicaciones con los permisos necesarios. El fallo estaba en el sistema de cifrado de estas conversaciones

-*Whatsapp es catalogada como una de las 41 aplicaciones que permiten el robo de datos.* [5] - Noviembre 2012.

En Noviembre de 2012, un grupo de investigadores alemanes hicieron un estudio sobre la aplicaciones móviles descargadas entre 39,5 y 185 millones de ocasiones con más fallos de seguridad. Whatsapp, la conocida aplicación de mensajería instantánea está entre ellas.

-*Los usuarios de Windows Phone llevan once días sin whatsapp* [12]

Además de los posibles fallos de seguridad, es común que estos servicios se caigan y dejen de funcionar durante algún tiempo. Los principales medios de comunicación se hacen eco de estos problemas:

Por tanto el sistema a desarrollar consiste en una aplicación de mensajería instantánea móvil, para dispositivos móviles con el sistema operativo Android, intentando mejorar los aspectos de privacidad de los datos de los usuarios, ya que es un aspecto muy solicitado por los usuarios y que se le ha prestado poca atención.

1.3. Alcance

Una de las principales tareas preliminares de cualquier proyecto software es la definición del alcance. Este proceso consiste en analizar la naturaleza del proyecto, y decidir qué funciones se incorporarán, y cuáles se quedarán fuera de este proyecto.

En nuestro proyecto (Talk Me) definimos el alcance de la siguiente manera:

Talk Me será una aplicación con una arquitectura Cliente-Servidor de mensajería instantánea, que sea capaz de comunicar a los usuarios que tengan instalada la aplicación.

Estos usuarios a través de la aplicación serán capaces de comunicarse mediante mensajes de texto, además de enviarse archivos desde cualquier parte del planeta con Internet, a través de su smartphone o tablet.

También debe ser capaz de crear grupos de conversación en los que los interlocutores se comunican con más de un receptor a la vez.

1.4. Objetivos

El sistema que se va a implementar consiste en una aplicación de mensajería instantánea para dispositivos móviles. Partimos de la existencia de distintas aplicaciones de este tipo, y nuestro trabajo se centrará en mejorar la privacidad de los usuarios al usarlas, asignatura pendiente de muchas de las aplicaciones existentes en el mercado.

El objetivo del trabajo será diseñar e implementar una aplicación completa que permita este tipo de comunicaciones de forma no intrusiva en la confidencialidad del usuario. Una aplicación con características similares, pero más respetuoso con la intimidad de los usuarios y de sus datos.

El proyecto se puede dividir en dos partes bien diferenciadas: por un lado el desarrollo del servidor web que permitirá y gestionará estas comunicaciones; y por otro lado el cliente que se instalará en los diferentes terminales móviles (Figura 3: Diseño del proyecto)



Figura 3: Diseño del proyecto

El cliente de la aplicación se desarrollará en Java [15] para la plataforma Android [16]. Esta decisión está tomada en base a las facilidades ofrecidas por Android para los desarrolladores, poniendo a su disposición una serie de API's y entornos de desarrollo que facilitan en cierta medida su labor.

Para el servidor utilizaremos Java porque además de ser un lenguaje multiplataforma bastante documentado y compatible con sistemas de paso de mensajes estándar como JSON [17], se dispone de amplia experiencia en su utilización.

El producto final será una aplicación para el sistema operativo móvil Android que permitirá el envío/recepción de mensajes en tiempo real entre los usuarios, que evite algunos de los problemas de intrusismo en la privacidad de los usuarios que se han conectado anteriormente (Figura 4: Vista general de conexiones del proyecto).



Figura 4: Vista general de conexiones del proyecto

1.4.1. Objetivos específicos

En este apartado se detallan con más precisión los objetivos a conseguir en el presente TFG.

Para la especificación y elicitación de requisitos nos apoyamos en el estándar IEEE 830-1998[18] y en técnicas de elicitación de requisitos tomadas de UML[19].

Nos apoyaremos en el estándar IEEE 830-1998, aunque no redactaremos el documento de forma estricta ya que sería inviable dadas las características del proyecto.

Como hemos explicado anteriormente, vamos a separar el proyecto en dos partes bien diferenciadas.

-Por un lado tendremos el servidor que gestionará las conexión y paso de mensajes entre unos usuarios y otros. A partir de ahora nos podremos referir a él como “el servidor”.

El servidor no almacenará información personal de los usuarios. Únicamente conocerá la información mínima e imprescindible para actuar de intermediario en el envío y recepción de mensajes entre usuarios.

-Por otro lado tenemos la aplicación móvil que estará instalada en cada uno de los dispositivos de los usuarios, y que será necesaria para acceder al servicio de mensajería. A partir de ahora nos podremos referir a él como “el cliente”.

La aplicación será capaz de enviar mensajes a otros usuarios de la aplicación sin dejar marcas de conexión como la última hora de conexión.

Los usuarios de la aplicación recibirán mensajes sin exponer información intrusiva al emisor, como la recepción del mensaje, o su estado en línea.

Se podrán crear grupos de conversaciones en los que habrá 3 ó más participantes.

En cuanto a los objetivos de aprendizaje, el presente trabajo nos permitirá abordar gran cantidad de tecnologías por lo que se aprenderá gran cantidad de conocimientos. A priori se pueden destacar los siguientes:

- Programación en Android
- Programación de un servidor Web utilizando tecnología JSP
- Servicios de Google para el desarrollo de herramientas Android, como GCM (Google Cloud Messaging) para comunicar el servidor con cada cliente de la aplicación.

1.5. Productos Similares

En la actualidad existe gran cantidad de aplicaciones de mensajería instantánea para smartphones, y tienen matices que los diferencian de las otras, aunque su funcionalidad básica es en todos la misma: crear un medio de comunicación entre personas mediante mensajes de texto a través de su smartphone o tablet y una conexión a internet.

Desde el lanzamiento de Android en 2008 por Google, han sido muchas las aplicaciones lanzadas por los desarrolladores para satisfacer esta necesidad. Unas con más éxito que otras, pero basadas siempre en la misma idea. A continuación se exponen las más relevantes:

 WhatsApp	<p>Whatsapp Messenger</p> <p>WhatsApp Inc.</p> <p>http://www.whatsapp.com/</p>
Información	Fue lanzado en 2009 y fue una de las primeras aplicaciones para este propósito. Cuenta con el mayor número de usuarios (500 millones [20]), y se estima que ha llegado a mover 54000 millones de mensajes al día. Hace pocos meses fue comprado por la compañía Facebook por un valor de 19000 millones de dólares.
Ventajas	Es una aplicación conocida y utilizada por la mayoría de los usuarios de Smartphone, por lo que usar esta aplicación permite comunicarte con la mayoría de usuarios de estos.
Desventajas	<ul style="list-style-type: none">-Pese a lo que parezca, es un software de pago. Aunque tenga una versión de prueba de 365 días, al finalizar este plazo, la aplicación se bloquea obligando al usuario que quiera seguir utilizando a pagar una cuota de 1\$ al año.-Whatsapp ha sufrido repentinos fallos en su conectividad, muchos ellas muy mediáticos, llegando a privar del servicio a los usuarios en algunos casos durante horas.-Poca privacidad para el usuario: Aunque poco a poco van mejorando en este aspecto, uno de las características de whatsapp es su falta de privacidad. Cualquier usuario puede saber la hora en que el otro usuario estuvo conectado al servicio por última vez, así como si ha recibido tus mensajes o no.-No está disponible una versión para ordenadores, ya sean Windows, MacOS o Linux-No permite llamadas de voz ni videoconferencias

Figura 5: Características WhatsApp

	<p>Line</p> <p>Naver Japan Corp.</p> <p>http://line.me/es/</p>
Información	<p>Es el principal rival de whatsapp en cuanto a número de usuarios. Cuenta con 300 millones de usuarios [21] y supera a Whatsapp en países como Japón y Corea. Posee una interfaz mucho más dinámica que whatsapp y destaca por la posibilidad de enviar “stickers” (Imágenes en movimiento).</p>
Ventajas	<ul style="list-style-type: none"> -Tiene una interfaz gráfica muy amigable y dinámica -Envío de stickers, lo que hace la aplicación más atractiva -Permite la realización de llamadas de voz y de video a través de las redes 3G, 4G y WIFI. -Dispone de un cliente de escritorio, por lo que podemos instalar la aplicación en cualquier ordenador portátil y de sobremesa y comunicarnos con nuestros contactos a través de su smartphone. -Dispone de servicio de atención al cliente disponible directamente desde la propia aplicación, como si de un contacto más se tratase.
Desventajas	<ul style="list-style-type: none"> -Tiene un uso excesivo de batería, posiblemente debido a su animada interfaz gráfica. -En España no es muy elevado el número de usuarios que lo tienen instalado, por lo que usarlo de forma exclusiva es una opción muy restrictiva.

Figura 6: Características Line

	<p>Viber</p> <p>Viber Media Inc.</p> <p><i>www.viber.com</i></p>
<p>Información</p>	<p>Es otro cliente de mensajería instantánea muy extendido. Pese a sus 200 millones de usuarios activos, posee prácticamente las misma funciones que los anteriores: llamadas, envío de mensajes etc. Sin embargo algunos medios recomiendan no usarlo ya que viola los derechos a la privacidad de sus usuarios, almacenando información suya y de sus contactos en sus servidores.</p>
<p>Ventajas</p>	<ul style="list-style-type: none"> -Permite llamadas y videollamadas a través de redes 3G, 4G y WIFI. -Es completamente gratuito, de hecho su licencia es de tipo “freeware” -Sincronización automática con la agenda del sistema. El usuario no tiene que preocuparse de agregar contactos al programa.
<p>Desventajas</p>	<ul style="list-style-type: none"> -Privacidad de datos del usuario: Viber ha sido muy criticado por el hecho de almacenar en sus servidores, datos del usuario y de sus contactos. -A pesar de disponer de funcionalidad para hacer videollamadas, esta se encuentra aún en fase Beta.

Figura 7: Características Viber

	<p>Telegram</p> <p>Telegram LLC</p> <p>www.telegram.org</p>
Información	<p>Una de las últimas aplicaciones de mensajería instantánea que han triunfado. Se presentó en agosto de 2013, y empezó a crecer en España en Febrero de 2014. Su éxito se basa principalmente en su enfoque a la seguridad de los datos. Actualmente cuenta con 35 millones de usuarios en todo el mundo.</p>
Ventajas	<ul style="list-style-type: none"> -Conversaciones seguras: Telegram nos permite realizar conversaciones seguras, que no se almacenan en sus servidores y solo conocen el emisor y el receptor. -Interfaz sencilla y poco cargada -Permite la autodestrucción de mensajes -Permite enviar archivos de hasta 1 GB. -El cliente de la aplicación es de código libre, lo que ha permitido la implementación de clientes alternativos y mejoras en general.
Desventajas	<ul style="list-style-type: none"> -No aporta nada novedoso a los anteriores. -Su servidor es código propietario, por lo que no se puede confirmar el nivel de seguridad del que presumen.

Figura 8: Características Telegram

2. Tecnología

En este capítulo se expondrán detalladamente las características de las tecnologías utilizadas para el desarrollo del sistema, las razones de su elección y posibles alternativas.

Para empezar recalcamos que el sistema comprende dos partes bien diferenciadas, el servidor y el cliente [1.4.1.], que se desarrollarán de forma independiente.

2.1. Servidor

En primer lugar, para el desarrollo del servidor del sistema, se ha elegido la tecnología JavaServer Pages (**JSP**). Las razones para su elección son las siguientes:

- El lenguaje en el que se programa esta tecnología es JAVA, lenguaje con el que se está muy familiarizado y agilizará el proceso de desarrollo.
- La tecnología JAVA hace que sea compatible con la mayoría de entornos, y así se desarrolla un servidor muy portable
- Es funcionalmente similar a otras tecnologías como PHP, con la ventaja de estar escrito en Java, lo que nos facilitará el trabajo.

2.1.1. Características

A continuación se detallarán las características de la tecnología seleccionada para el desarrollo del servidor del sistema (JSP).

Java Server Pages (JSP) es una tecnología para el desarrollo de páginas web dinámicas basadas en html y xml utilizando Java como lenguaje de programación y desarrollado por “Sun Microsystems”, actualmente en posesión de “Oracle”. Funcionalmente, JSP es similar a PHP pero usando el lenguaje Java.

La última versión estable es la 2.1

Para la ejecución de páginas JSP se necesita de un servidor web, compatible con Servlets de Java. Por ejemplo “Apache”.

JSP como cualquier otro lenguaje tiene muchas similitudes con otros lenguajes de programación, y tiene ciertas ventajas (e inconvenientes) frente a estos que nos harán decidir entre un lenguaje u otro.

- ASP.net : Al ser ASP una tecnología de Microsoft, está escrita en Visual Basic, por lo cual JSP al estar escrito en Java cuenta con la portabilidad de Java.

- Servlets: En principio JSP hace exactamente lo mismo que los Servlets de Java, pero es mucho más cómodo programar en JSP que en Servlets ya que en los Servlets deberíamos hacer un “print” para cada línea que queremos que se muestre en la página; sin embargo JSP se embebe directamente en el HTML. Además con JSP podemos separar fácilmente el formato del contenido de la página. Es decir, el diseñador web puede diseñar toda la página dejando en blanco los huecos que el programador Java cubrirá. Se podría decir que las páginas JSP son una forma alternativa y cómoda de programar Servlets, ya que la página JSP antes de transformarse en HTML se transforma en un Servlet.
- PHP: En realidad PHP y JSP son muy parecidos. Una de las ventajas de usar JSP frente a PHP es el uso del lenguaje Java, que posee gran cantidad de documentación y está muy extendido. Un aspecto en contra de JSP frente a PHP es que el hosting en JSP suele ser más caro que en PHP.

A continuación se muestra un ejemplo de una pequeña página en JSP (Figura 9)

```

1  <html>
2      <head>
3          <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
4          <title>Ejemplo JSP</title>
5      </head>
6      <body>
7          <H1> </H1>
8          <%= new java.util.Date().toString() %><br>
9          <%-- Convertir a mayúsculas un String --%>
10         <%= "Pasar a mayúsculas".toUpperCase() %><br>
11         <%-- Resultado de una expresión aritmética --%>
12         <%= (5+2)/(float)3 %><br>
13         <%= new java.util.Random().nextInt(100) %>
14     </body>
15 </html>
16

```

Figura 9: Ejemplo JSP

Como se ve en el ejemplo, el código va escrito en html y para incluir sentencias java solo hay que ponerlas entre <% %>.

El servidor lee esta página jsp y genera una página en html para el navegador del cliente. (Figura 11). Este proceso se ilustra de forma gráfica en la Figura 10.

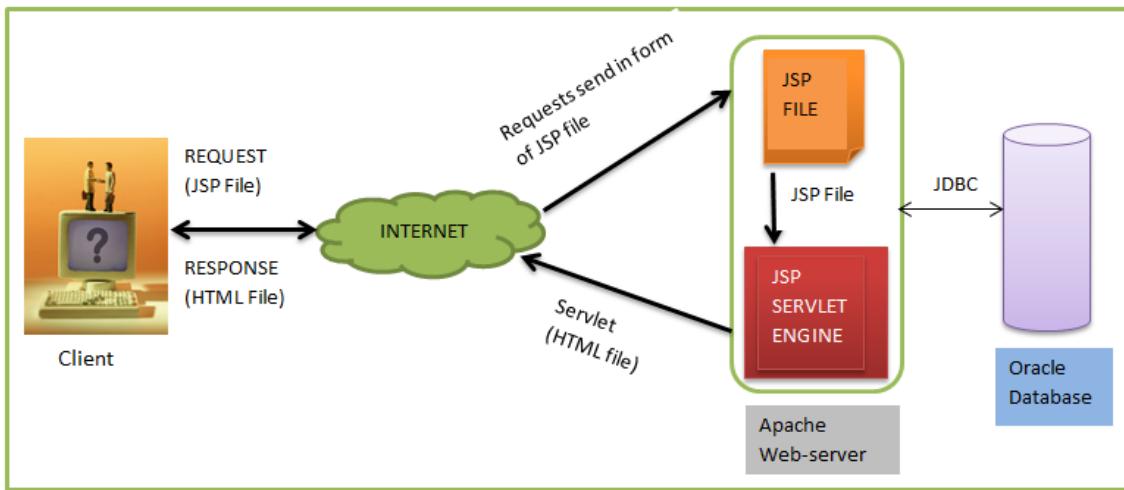


Figura 10: Proceso de generación de página JSP

El cliente no tiene acceso al código java, sino solo al html que genera.

Para el ejemplo anterior, la página html resultante es:

```

1 <html>
2   <head>
3     <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
4     <title>Ejemplo JSP</title>
5   </head>
6   <body>
7     <H1> </H1>
8     Wed Jan 16 19:00:52 CET 2013<br>
9     PASAR A MAYÚSCULAS<br>
10    2.3333333<br>
11    65
12  </body>
13 </html>
14

```

Figura 11: HTML generado con JSP

Como podemos apreciar, en el navegador del cliente no hay rastro del código java, tan solo los resultados que queremos mostrar en la página.

2.1.2. Sintaxis

JSP insistimos, está basado en HTML y JAVA, pero tiene ciertas peculiaridades que a la vez es lo que lo caracterizan y lo hacen competente para la generación de contenido web dinámico.

Variables Implícitas

Son variables que están declaradas de forma predeterminada y podemos acceder a ellas de forma directa. Estas variables nos brindan gran cantidad de información. Son las siguientes:

- **pageContent:** Nos da acceso a los objetos y atributos de la página JSP. javax.servlet.jsp.PageContext
- **request:** Sirve para acceder a los datos que nos han mandado los usuarios en su petición HTTP. javax.servlet.http.HttpServletRequest
- **response:** Determina los datos que serán devueltos al usuario en la respuesta HTTP. javax.servlet.http.HttpServletResponse
- **Sesión:** Nos permite guardar información de cada usuario que ha accedido a la página JSP. javax.servlet.http.HttpSession
- **config:** Sirve para pasar información del archivo web.xml a los servlets. javax.servlet.ServletConfig
- **application:** nos permite guardar información de toda la aplicación web. javax.servlet.ServletContext
- **out:** Salida estándar de la página JSP, es decir en el cuerpo del HTML. javax.servlet.jsp.JspWriter
- **page:** Datos de la página JSP. java.lang.Object
- **exception:** nos permite acceder la excepción ocurrida. Solo puede utilizarse en las páginas que están marcadas como isErrorPage

Directivas

Son etiquetas que configuran la página JSP. No ofrecen información al usuario, sino al motor JSP. Son las siguientes:

- **Include:** incluye el contenido de un fichero en la página jsp. Ejemplo:

```
<%@ include file="header.html" %>
```

- **Taglib:** como su propio nombre indica, importa librerías de etiquetas. Ejemplo:

```
<%@ taglib uri="/META-INF/taglib.tld" prefix="str" %>
```

Donde:

- **Uri:** permite localizar el fichero descriptor de la librería.
 - **Prefix:** especifica el identificador que todas las etiquetas de la librería deben incorporar
- **Page:** Modifica atributos de la página JSP a procesar (algunos son variables implícitas y tienen ya un valor por defecto). Estos atributos son:
 - **Import:** sirve para importar paquetes java y utilizarlos dentro de la página JSP. Funcionalmente es el clásico import de Java. Ejemplo:

```
<%@ page import ="Dominio.GestorUsuario"%>
```

Donde GestorUsuario es una clase de Java contenida en el paquete Dominio.

- **Sesión:** especifica si se van a utilizar los datos contenidos en la sesión (variable implícita "sesión"). Por defecto está asignado un "true". Ejemplo:

```
<%@ page session="false" %>
```

- **ContentType:** Especifica el tipo MIME(protocolo para el intercambio de archivos) del objeto "response" (varibale implícita descrita anteriormente). Por defecto es "text/html; charset=ISO-8859-1". Ejemplo:

```
<%@ page contentType="class; class" %>
```

- **Buffer:** Tamaño del buffer de salida de la variable implícita “out”. Por defecto es 8KB, pero puede ser incluso 0 indicandole el valor “none”. Ejemplo:

```
<%@ page buffer="24KB" %>
```

- **errorPage:** Indica la ruta de la página de error que se mostrará en caso de que se produzca una excepción en la ejecución de la página JSP. Ejemplo:

```
<%@ page errorPage="/error/error_page_login" %>
```

- **isErrorPage:** Indica si la página en la que está siendo invocado es una página JSP que maneja errores. Puede tomar valores “true” o “false”. Únicamente las páginas con este valor a true podrán ser accedidas con la variable implícita “exception”. Ejemplo:

```
<%@ page isErrorPage="true" %>
```

Declaraciones

La declaración de variables y métodos se hace de forma similar a como lo haríamos en Java. Como es un fragmento de Java irá entre “`<% %>`”, pero con la particularidad de que va precedido del símbolo “!”.

Por ejemplo:

```
<%! int contador = 0; %>
```

Las variables y métodos inicializados serán globales en toda la página. Como el motor JSP transformará la página en un Servlet, tenemos que tener en cuenta que la página tendrá numerosas peticiones y esta variable tendrá el mismo valor para todas las peticiones.

Scriptlets y expresiones

Los scriplets son código Java incrustado entre los elementos de la página.

Las expresiones se evalúan dentro del motor JSP y solo se muestra el resultado. No deben acabar en “;”.

```
1  <HTML>
2  @@
3      String titulo = "";
4      if (request.getAttribute("titulo") != null) {
5          titulo = (String) request.getAttribute ("Prueba expresion");
6      }
7  @@
8  ...
9  <title><%=titulo%></title>
10 ....
11 </HTML>
12 |
```

Amarillo: Scriptlets

Azul: expresión

Etiquetas JSP

Son proporcionadas por JSP y nos proporcionan una funcionalidad básica.

- <jsp:forward> : redirige la petición a otra página.
- <jsp:include> Incluye el texto de un fichero externo dentro de la misma página.
- <jsp:plugin>: se usa para mostrar un objeto, normalmente un applet o un Bean en el explorador del cliente.
 - o type: el tipo de objeto que vamos a incluir en el plug-in.
Pueder tomar valores “vean” o “applet”
 - o code: nombre de la clase java que contiene el applet o bean.
 - o Codebase: ruta de la clase java que ejecutará el plug-in

También existen etiquetas destinadas exclusivamente a manipular componentes JavaBean:

- **<jsp:useBean>** : nos permite manipular un Bean, y si no existe crearlo, especificando su ámbito, clase y tipo.
- **<jsp:getProperty>**: obtiene las propiedades del Bean, y las guarda en el objeto response de la página jsp.
- **<jsp:setProperty>**: establece las propiedades de un bean.

Etiquetas JSTL

Son etiquetas proporcionadas por la compañía SUN, y extienden de las etiquetas jsp. Las más importantes son:

- **core**: iteraciones, condicionales, manipulación de URL's y otras funciones generales.
- **xml**: permite manipular ficheros XML y XML-Transformation
- **sql**: permite gestionar la conexión un la página con una base de datos sql
- **i18n**: funciones de internacionalización y formato de cadenas

Struts Tag Libraries

Están distribuidas por la compañía “Apache” y funcionan junto el Framework de Struts.

El framework Struts proporciona un conjunto de 6 librerías de etiquetas, que asisten en la tarea de la creación de la vista de MVC para evitar incluir código Java en los JSPs. Son las siguientes:

- **Bean Tags**
- **HTML Tags**: Se usan principalmente para crear formularios de entrada de datos y funcionalidades de interface basadas en HTML. Sus acciones más comunes son:

- o **Base**: genera un elemento HTML <base>
 - o **Errors**: visualiza un conjunto acumulado de mensajes de error.
 - o **Form**: define un formulario de entrada de datos
 - o **Text**: define un campo de entrada de texto
 - o **Messages**: almacena conjunto de mensajes acumulados
 - o **Submit**: visualiza un botón de entrega.
- **Logic Tags**: tiene utilidades para la iteración de colecciones, generación condicional de una saluda y control del flujo de aplicación. Sus acciones mas comunes son:
 - o **Present**: genera el contenido de marcado dentro de esta etiqueta si el valor indicado es encontrado en esta petición
 - o **notPresent**: lo opuesto a present
 - o **itérate**: repite el contenido anidado dentro de esta etiqueta al iterar sobre una colección
 - o **forward**: transfiere control a la página especificada por la entrada ActionForward. Nested Tags
- **Template Tags**: Un template es una página JSP que usa una librería de etiquetas personalizadas para describir la disposición (layout) de los componentes de la página

Define cómo aparecerán visualmente las páginas JSP de una aplicación, sin especificar el contenido. Esta definición es utilizada por todos los JSPs sin necesidad de mezclar layout con el contenido.
- **Tiles Tags**: La librería de etiquetas Tiles es un super-conjunto de la librería Templates. Intenta evitar la duplicación de contenido de lenguaje de marcado dentro de una aplicación web con respecto al look-and-feel de un portal. Tiles reduce el código redundante en una aplicación web y separa el contenido de la visualización del mismo de manera más eficiente.

2.1.3. Rendimiento

El estudio de CppCMS Blog [29] nos dejó la siguiente comparativa entre las distintas tecnologías de servidores (Figura 12: Rendimiento JSP):

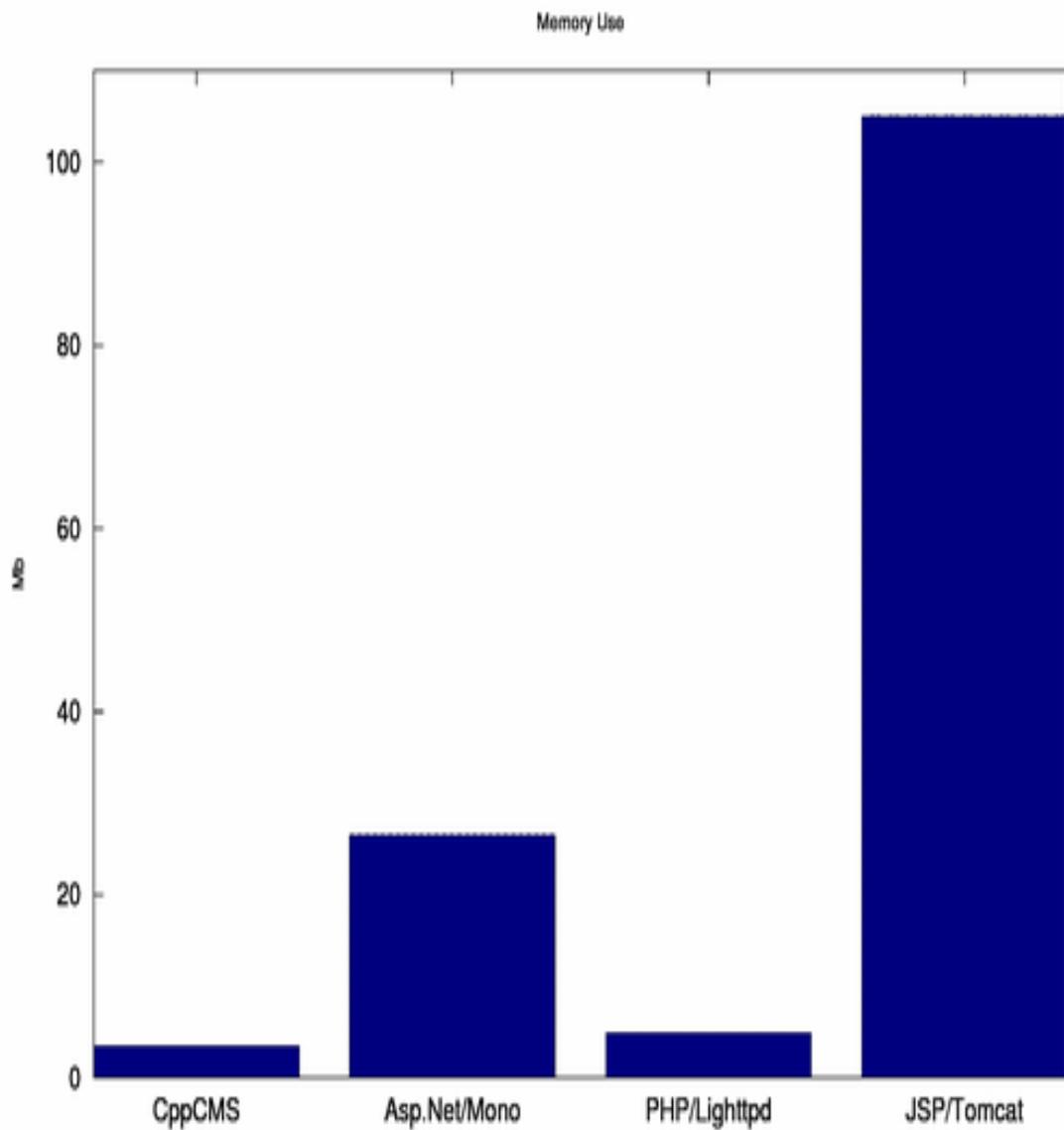


Figura 12: Rendimiento JSP

Una de las desventajas del uso de Java (JSP) es el consumo de memoria. Al utilizar páginas JSP este uso de memoria también se dispara con respecto a otras tecnologías como asp.Net y PHP.

En cuanto a las oportunidades laborales, PHP tiene una gran ventaja frente a asp y jsp, que está en un discreto tercer lugar (Figura 13).

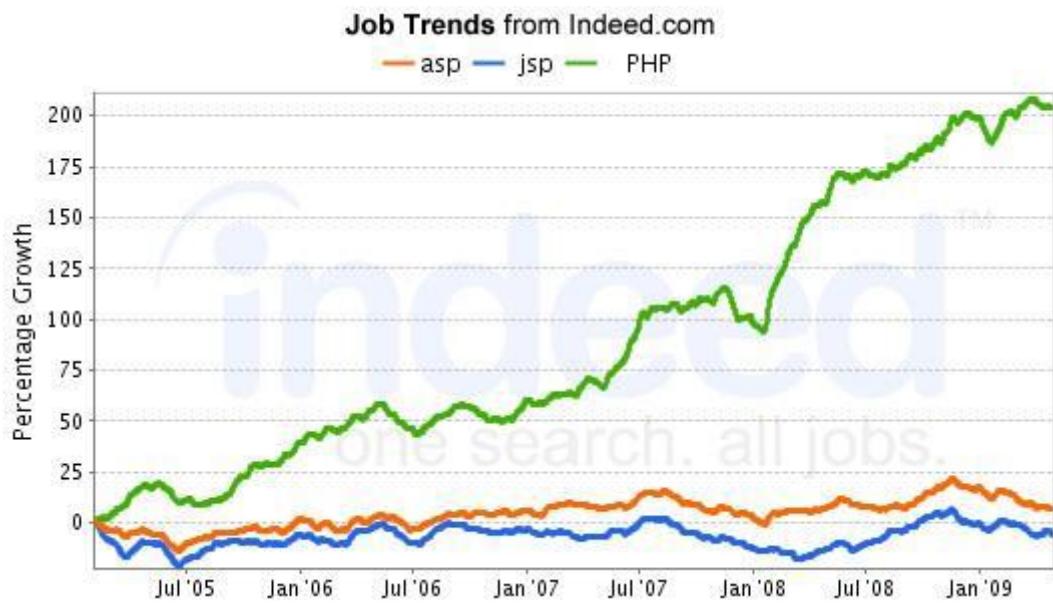


Figura 13: Puestos de trabajo JSP

El siguiente gráfico (Figura 14) compara diferentes aspectos de la tecnología JSP y PHP. Destaca el elevado coste y la elevada formación necesaria para programar en JSP frente a PHP, aunque también es mucho más modular. El rendimiento a pesar de que JSP consume muchos más recursos, es similar en ambas tecnologías.

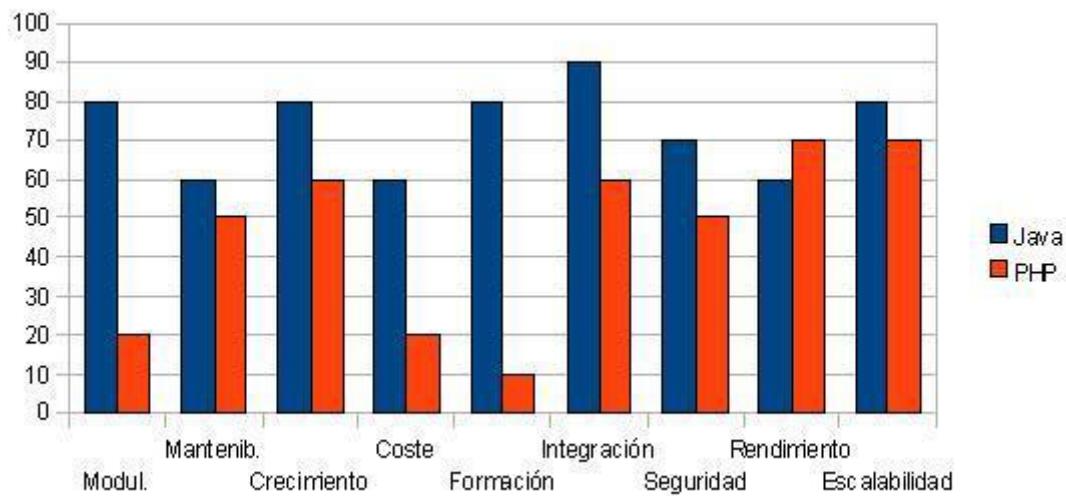


Figura 14: JSP vs PHP

2.1.4. Resumen y conclusiones

- JSP es una tecnología de programación de páginas web con contenido dinámico
- JSP permite generar código HTML mediante el procesamiento de código Java en el servidor.
- JSP es un fichero HTML con código Java en su interior. Los Servlets son un programa en Java que genera un HTML mediante secuencias de “println”.
- JSP es una forma cómoda de programar Servlets
- JSP es equivalente funcionalmente a PHP.
- JSP es un lenguaje de servidor.
- PAGINA JSP --> SERVLET --> PAGINA HTML
- El usuario no tiene acceso al código java. Este se compila en el servidor y al cliente se le devuelve la página HTML resultante.

2.2. Cliente

La elección del sistema operativo del cliente ha sido algo más clara, debido a los limitados sistemas operativos móviles existentes, que el servidor. Nos hemos decantado por desarrollar la aplicación para sistemas operativos Android frente a hacerlo para iOS o Windows Phone. Las razones para esta elección son las siguientes:

En primer lugar, el número de usuarios. Según el *IDC 2013 [2]*, el número de terminales vendidas con el sistema operativo Android, supera los 1000 millones. Una aplicación desarrollada para este sistema operativo estaría disponible para más de 1000 millones de usuarios. Estos datos no tienen intención de reducirse, y mantienen su ventaja respecto a su principal competidor (iOS), en un alejado segundo lugar.

En abril de 2013 se hizo público que Android alcanzó el 92% en ventas de nuevos *smartphones* para el trimestre comprendido entre diciembre 2012 y febrero 2013 en España, seguido de iOS con un 4.4% [22]

Operating System	3Q12 Shipment Volumes	3Q12 Market Share	3Q11 Shipment Volumes	3Q11 Market Share	Year-Over-Year Change
Android	136.0	75.0%	71.0	57.5%	91.5%
iOS	26.9	14.9%	17.1	13.8%	57.3%
BlackBerry	7.7	4.3%	11.8	9.5%	-34.7%
Symbian	4.1	2.3%	18.1	14.6%	-77.3%
Windows Phone 7/ Windows Mobile	3.6	2.0%	1.5	1.2%	140.0%
Linux	2.8	1.5%	4.1	3.3%	-31.7%
Others	0.0	0.0%	0.1	0.1%	-100.0%
Totals	181.1	100.0%	123.7	100.0%	46.4%

Figura 15: Ventas de smartphones 3Q 2013. Fuente: IDC 2013[2]

Otro aspecto a tener en cuenta, es que el lenguaje de programación de Android es Java, mientras que en iOS es Objective-C [23], lo cual decanta la línea de aprendizaje en favor de Android.

Otro factor decisivo en este caso ha sido la licencia. Android, el sistema operativo de Google, es software libre [24], lo cual nos ofrece muchas facilidades a la hora de desarrollar en esta tecnología. Sin embargo iOS, es un sistema operativo privativo de Apple, y para desarrollar para este necesitaríamos unos requisitos que supondrían un gasto económico.

2.2.1. Características

Las aplicaciones de Android están programadas en Java. Cuando desarrollamos una aplicación con el SDK de Android, este nos genera un archivo tipo *.apk. Este archivo contiene todo lo necesario para instalar la aplicación en cualquier dispositivo con sistema operativo Android.

Componentes:

Una aplicación Android está compuesta de varios tipos de componentes, los cuales no tienen por qué interactuar directamente con el usuario. Aunque muchos componentes dependan unos de otros, todos existen como entidades independientes y tienen una función concreta.

Existen cuatro tipos diferentes de componentes, y cada uno tiene una función y un ciclo de vida distinto dentro de la aplicación: Activities, Services, Content providers y Broadcast Receivers.

- **Activities:** Una actividad (Activity) representa una interfaz de usuario. Una aplicación puede tener varias activities, y aunque trabajen juntas para generar las acciones del usuario, las activities son independientes entre sí.
- **Services:** Un servicio (Service) es un componente que se ejecuta en segundo plano, realizando tareas de larga duración o remotas con el fin de no bloquear la interacción del usuario con el activity mientras se completan. Los servicios no proporcionan interfaz gráfica.
- **Content Providers:** Un Content Provider es un componente que nos permite compartir información de la aplicación con otras aplicaciones. Una aplicación que desee compartir ciertos datos con otras aplicaciones deberá implementar un Content Provider. El propio sistema Android implementa un Content Provider que permite a las aplicaciones acceder a la agenda, la cámara etc.
- **Broadcast Receivers:** Un Broadcast Receiver es un componente de Android que nos permite registrar eventos del sistema. Es un sistema que sirve para comunicar aplicaciones entre sí, capturando los eventos con *onReceive()* o enviándolos al sistema con *sendBroadcast()*

Un aspecto importante del sistema Android es que los componentes de las aplicaciones son independientes, y una aplicación puede llamar al componente de otra aplicación (por ejemplo la función cámara) sin necesidad de ejecutar la aplicación correspondiente. Esto se realiza mediante peticiones al sistema a través de un objeto “Intent”.

El archivo Manifest:

El archivo manifest.xml es un fichero localizado en la raíz del proyecto Android donde se configuran las opciones y propiedades del proyecto. Es un archivo xml estructurado, que debe ser modificado con cada versión publicada de la aplicación. Su principal contenido es:

- Declarar los componentes de la aplicación
- Declarar el mínimo nivel API requerida por la aplicación, es decir, la versión mínima de Android en el que la aplicación funcionará.
- Declarar características de hardware y software utilizados o requeridos por la aplicación, tales como una cámara, los servicios Bluetooth o una pantalla multitouch.
- Las bibliotecas utilizadas en el sistema (excepto las propias de Android), por ejemplo la biblioteca de Google Maps .
- Definir los permisos que la aplicación requiere de Android, como el acceso a Internet o acceso de lectura a los contactos del usuario.

Recursos de la Aplicación

Además del código fuente, una aplicación Android está compuesto de otros recursos, como imágenes, ficheros, y archivos xml de configuración de interfaces (layouts). Todo este contenido extra se encuentra en la carpeta /res en la raíz del proyecto. La estructura de este directorio es la siguiente (Figura 16):

Carpeta	Descripción
/res/drawable/	<p>Contiene las imágenes usados en la aplicación. En función de la densidad de estas, se separan en varias carpetas:</p> <ul style="list-style-type: none"> • /drawable-ldpi (densidad baja) • /drawable-mdpi (densidad media) • /drawable-hdpi (densidad alta) • /drawable-xhdpi (densidad muy alta) <p>La idea es tener una imagen similar en cada una de las carpetas, y que sea el propio sistema el que decida cuál usar en función de la resolución de la aplicación.</p>
/res/layout/	Contiene los ficheros XML de configuración de la interfaz gráfica de usuario de cada activity.
/res/anim/ /res/animator/	Definen las animaciones que se utilizan en la aplicación
/res/color/	Contiene ficheros XML en el cual se definen colores para utilizar en la aplicación
/res/menu/	Contienen ficheros XML los cuales definen los menús de la aplicación
/res/xml/	Contiene ficheros XML utilizados por la aplicación que no coinciden con la descripción de las demás categorías.
/res/raw/	Contiene recursos que no son del tipo XML, los cuales no coinciden con la descripción de las demás categorías.
/res/values/	Contiene otros fichero XML de recursos, como colores, cadenas y estilos.

Figura 16: Estructura de directorios de un proyecto Android

Uno de los aspectos más importantes de separar los recursos del código, es la posibilidad de configurar distintos recursos en función del dispositivo donde se encuentre instalada la aplicación, prácticamente de manera automática. Por ejemplo: los recursos incluidos en la carpeta “values-v11” se aplicarían tan sólo a dispositivos cuya versión de Android sea la 3.0 (API 11) o superior. Al igual que el sufijo “-v” existen otros muchos para referirse a otras características del terminal.

2.2.2. Arquitectura

Para comprender como funciona Android, es indispensable comprender su arquitectura y por qué está diseñada así.

En las siguientes líneas se dará una visión global por capas de cuál es la arquitectura empleada en Android. Cada una de estas capas utiliza servicios ofrecidos por las anteriores, y ofrece a su vez los suyos propios a las capas de niveles superiores, tal como muestra la siguiente figura

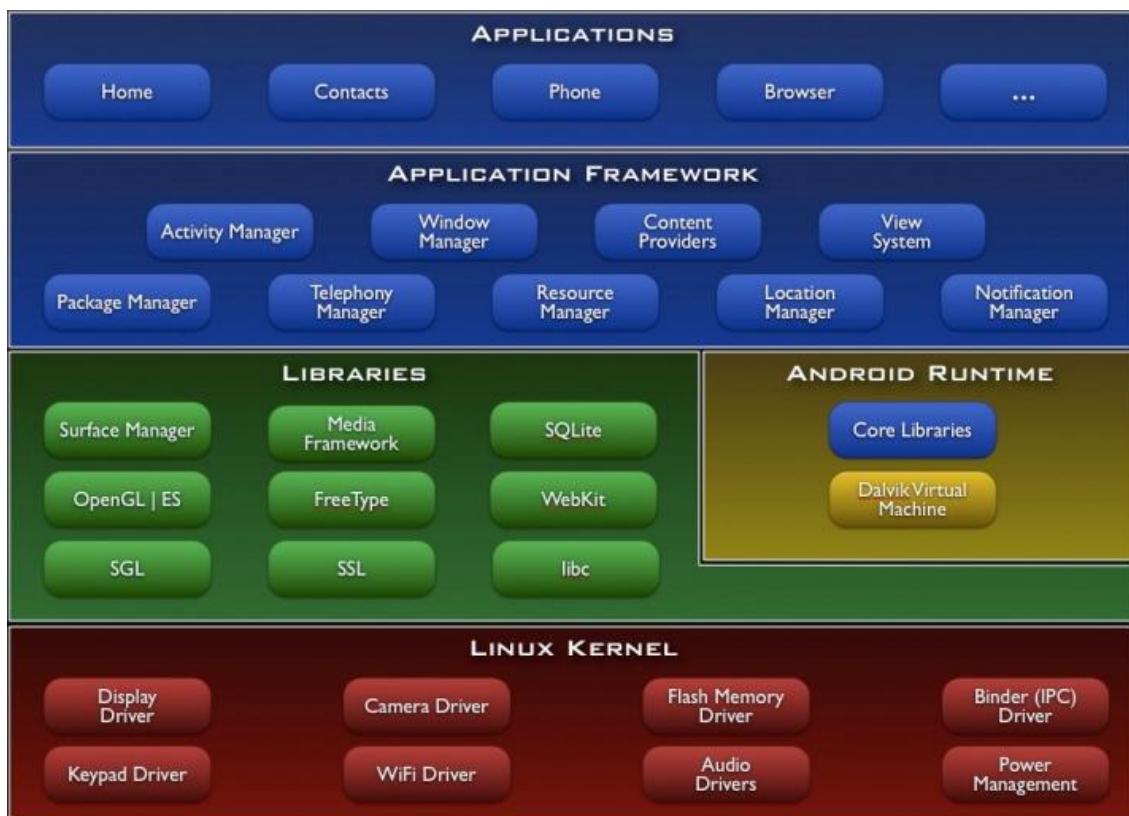


Figura 17: Arquitectura Android

La arquitectura del Sistema Operativo Android está formada por 4 capas, las cuales reciben servicios de las capas inferiores, y proveen de servicios a las superiores. A continuación se describen las capas detalladamente:

- 1. Linux Kernel:** El núcleo de Android está basado en el kernel de Linux [25], más concretamente en la versión 2.6., como cualquier distribución de Linux, como Ubuntu. Android utiliza esta capa para los servicios base del sistema como como gestión de memoria y de procesos, pila de red, modelo de controladores y seguridad. Además el núcleo también actúa como una capa de abstracción en el hardware del dispositivo y el resto de la pila de software. Esto permite acceder a los componentes del dispositivo sin necesidad de conocer sus características, lo cual facilita el trabajo del desarrollador. Al Kernel de Linux, se han añadido una serie de mejoras para sacarle el mayor rendimiento a un dispositivo móvil: gestión de memoria, gestión de energía, etc.
- 2. Libraries (Bibliotecas):** Es la capa situada justo encima del Kernel. Está formada por un conjunto de bibliotecas de C/C++. Existen gran cantidad de librerías en esta capa y las más importantes y comunes son las siguientes:
 - **Surface Manager (Gestor de superficies):** Se encarga de componer las imágenes que se muestran por la pantalla del terminal a partir de capas 2D y 3D. Cuando una aplicación intenta mostrar algo por pantalla, usa esta librería, que en vez de mostrarla directamente por la pantalla, modifica las imágenes, las almacena en memoria y finalmente combina estas imágenes para mostrarlas en la pantalla. Esto ofrece al desarrollador gran cantidad posibles efectos: superposición de elementos, transparencias, animaciones...
 - **SGL (Scalable Graphics Library):** Es el motor gráfico 2D de Android. Fue desarrollada por Skia, aunque en 2005 fue adquirida por Google. Se encarga de representar los elementos en dos dimensiones. Además de en Android, Google, también usa esta librería en su explorador Chrome.
 - **OpenGL ES (OpenGL for Embedded Systems):** Es el motor gráfico 3D de Android. Utiliza aceleración hardware o un motor software altamente optimizado cuando el dispositivo no la proporciona.
 - **Media Framework (Bibliotecas multimedia):** Son un conjunto de librerías que están basadas en OpenCore. Proporcionan las características necesarias para visualizar, reproducir o grabar numeros formatos de imagen video y audio como JPG, GIF, PNG, MPEG4, AVC (H-264), MP3, AAC, o ARM.
 - **FreeType:** Permite mostrar fuentes tipográficas, tanto basadas en mapas de bits como vectoriales [26]. Su objetivo es acceder a los tipos de letra de forma sencilla y convertirlas en mapas de bits.
 - **SSL (Secure Sockets Layer):** Es una biblioteca que proporciona seguridad al acceder a Internet, utilizando medidas de seguridad basadas en criptografía.

- **SQLite**: Es el motor de bases de datos disponible para todas las aplicaciones del terminal.
 - **WebKit**: Es el motor web que utiliza tanto el navegador del sistema, como el navegador web embebido en otras aplicaciones. Es el mismo motor utilizado en Google Chrome y Safari.
 - **Libc**: Es una biblioteca escrita en C que proporciona la funcionalidad básica para la ejecución de las aplicaciones. Esta basada en la implementación Berkeley Software Distribution (BSD), pero optimizado para sistemas Linux embebidos.
3. **Runtime de Android**: Está en el mismo nivel que las bibliotecas de Android ya que también está formado por librerías. Incluye un conjunto de librerías base que proporcionan la mayor parte de la funcionalidad de las bibliotecas de Java. Cada aplicación del sistema funciona en su propio proceso en su propia instancia de la máquina virtual Dalvik [27].

Dalvik es la máquina virtual usada por los dispositivos Android para ejecutar las aplicaciones. El proceso es similar al utilizado por la máquina virtual de Java, pero Dalvik usa un bytecode diferente. Esta máquina está diseñada para ejecutar varias instancias de sí misma simultáneamente. Como está diseñada específicamente para el sistema operativo Android, está optimizada para necesitar poca memoria, un factor decisivo en los dispositivos que utilizan este sistema operativo. Además se basa en registros en vez de en pilas, mejorando el rendimiento de los terminales.

4. **Application Framework**: Es un conjunto de APIs que nos ofrece Android, la mayoría bibliotecas de Java que acceden a los recursos a través de la máquina virtual Dalvik. Son las mismas librerías que las utilizadas por las aplicaciones base.

Como se aprecia, la arquitectura de Android está diseñada para simplificar la reutilización de componentes, ya que una aplicación publica sus capacidades, y otra puede hacer uso de las mismas (si lo tiene permitido).

Los elementos de esta capa más importantes son:

- **Activity Manager (Administrador de actividades)**: Se encarga de controlar el ciclo de vida de las activity y la pila de ejecución de estas.
- **Windows Manager (Administrador de ventanas)**: Se encarga de gestionar el contenido que aparece en pantalla, creando superficies que serán completadas con el contenido de las actividades.

- **Content Provider (Proveedor de contenidos):** Permite la comunicación y consulta y publicación de datos entre aplicaciones de forma segura y manteniendo el control de los datos que son accesibles por terceras aplicaciones. El mismo sistema Android nos ofrece unos Content Providers con información, por ejemplo el Content Provider que nos ofrece información de los contactos almacenados en el teléfono.
- **Views (Vistas):** Son los controles de usuario que se incluyen en las actividades, como los botones, los cuadros de texto y las listas, pero además existen muchos más, y algunos más sofisticados como el navegador web o un visor de Google Maps.
- **Notification Manager (Administrador de notificaciones):** Es el encargado de avisar al usuario cuando algo requiera su atención. Puede mostrar una alerta en la barra de notificación, pero también puede emitir un sonido, controlar el vibrador o los LED's del terminal.
- **Package Manager (Administrador de paquetes):** Controla la información de instalación de las aplicaciones, y la información de los paquetes .apk que los contienen.
- **Telephony Manager (Administrador de telefonía):** Proporciona acceso al hardware de telefonía del dispositivo. Nos permite realizar y recibir llamadas, sms y MMS.
Una característica importante de este componente es que no permite la modificación ni eliminación de la actividad que se muestra durante una llamada en curso por motivos de seguridad.
- **Resource Manager (Administrador de recursos):** Proporciona acceso a todos los elementos de una aplicación que se incluyen en el código, como imágenes, cadenas, traducción a otros idiomas y layouts. Permite gestionar estos elementos fuera del código de la aplicación y generar diferentes versiones de esta en función de la resolución de pantalla, idioma, y otras características.
- **Location Manager (Administrador de ubicaciones):** Gestiona el uso del posicionamiento, tanto por GPS como por redes, y el trabajo con mapas y posiciones geográficas.
- **Sensor Manager (Administrador de sensores):** Nos permite gestionar todos los sensores de los que esté provisto nuestro terminal: los LEDs, el acelerómetro, brújula, sensor de proximidad, sensor de temperatura, de campo magnético, etc.

Applications (Aplicaciones): En esta última capa se encuentran tanto las aplicaciones base de Android (explorador web, mensajes, agenda etc), como las aplicaciones instaladas posteriormente en sistema operativo. También se encuentra en esta capa el launcher, que es una aplicación que nos permite lanzar las demás aplicaciones.

2.2.3. Compilación

Pese a lo que pueda parecer, las aplicaciones que ejecutamos en Android no se ejecutan en una máquina virtual de Java, por lo que a los ficheros generados por nuestro código fuente en Java, hay que aplicarle un proceso para convertirlos en ficheros android instalables (*apk*)

El proceso de transformación del código Java a una aplicación Android *.apk se detalla a continuación.

Partimos de un código fuente escrito en java. Estos ficheros son del tipo “*.java*”. Compilamos estos ficheros con el compilador de Java obteniendo los archivos binarios de Java “**.class*”.

Ahora estos ficheros binarios Java hay que transformarlos en ficheros binarios para la máquina virtual de Android (*.dex*). Este proceso lo lleva a cabo el compilador Dex.

Estos ficheros *.dex* son entendibles por la máquina virtual de Android Dalvik, así que esta los organiza y los empaqueta junto a los recursos utilizados en el código Java, generando así un archivo *.apk* instalable en cualquier dispositivo Android.

Este proceso se detalla en la sección de documentación técnica de la API de Android [37]. En las Figuras 18 y 19 se resume su proceso:

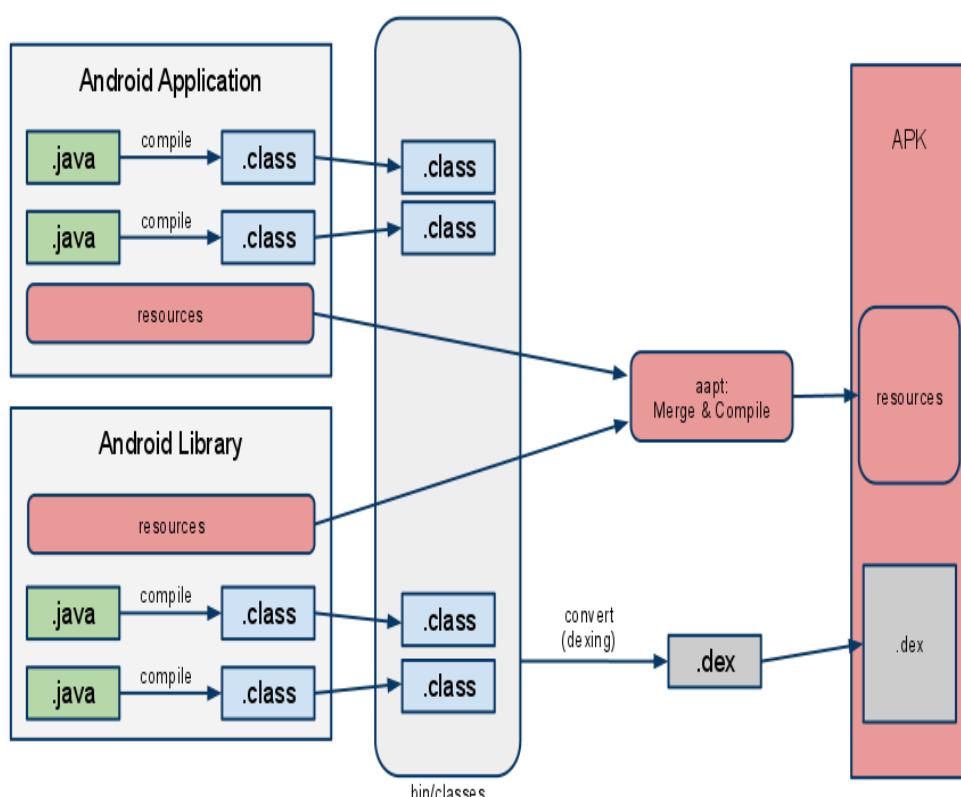


Figura 18: Proceso de compilación Android. <http://developer.android.com/tools/building/index.html>

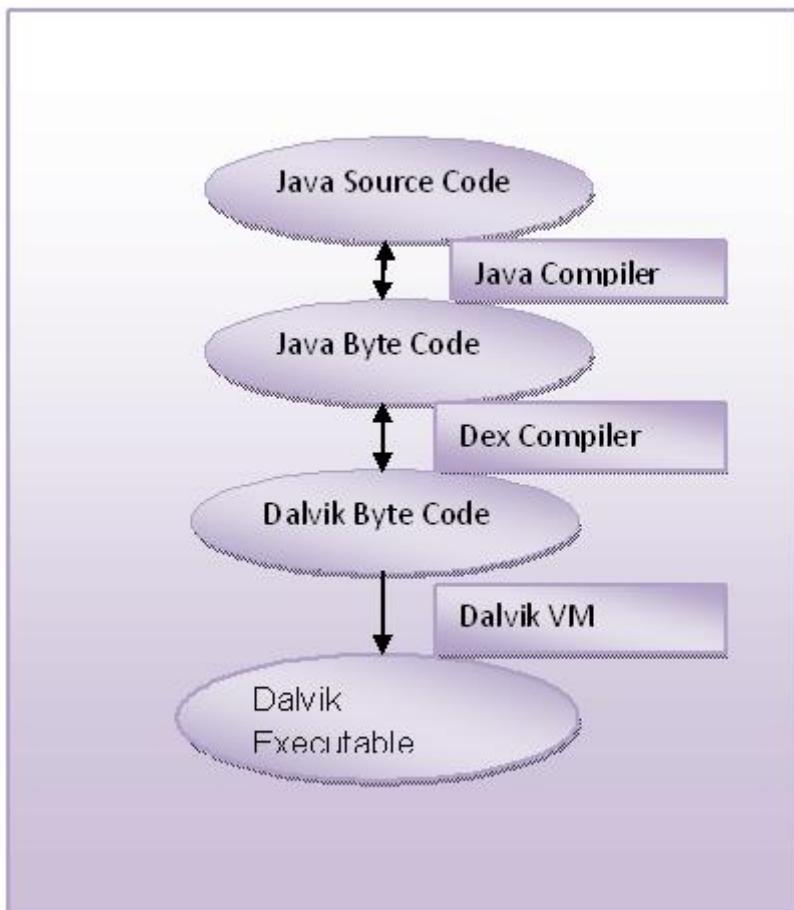


Figura 19: Proceso de compilación Android (2) <http://developer.android.com/tools/building/index.html>

3. Análisis del Sistema

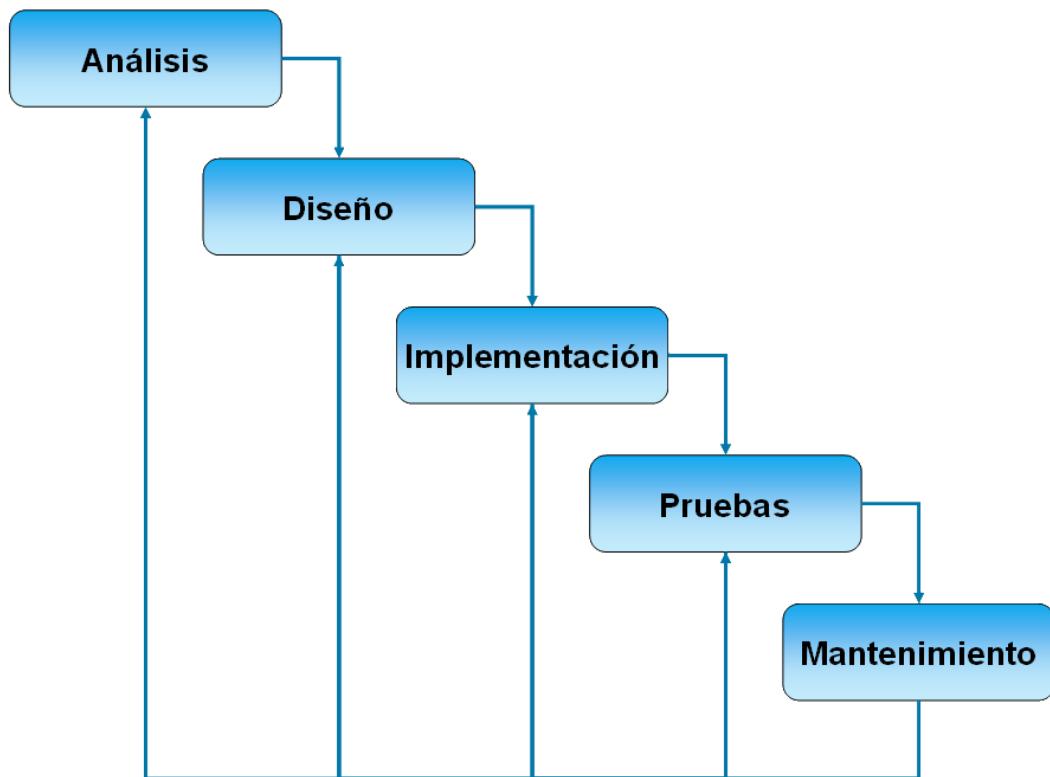


Figura 20: Ciclo de vida clásico

Según el ciclo de vida clásico de un programa (Figura 20), el primer paso en cualquier desarrollo software es un análisis del sistema que se va a implementar.

Puede que actualmente el ciclo de vida clásico no sea el mejor modelo de referencia, aunque todos los modelos coinciden en algo con él: hay que realizar una etapa de análisis inicial previa a todo desarrollo software.

En nuestro caso no iba a ser menos, y se han analizado las necesidades de nuestro proyecto, analizando los casos de uso, y generando unos requisitos a partir de ellos.

3.1. Casos de Uso

Los casos de uso son una técnica para capturar los requisitos de usuario de la aplicación a desarrollar. Son realmente útiles cuando el usuario no tiene conocimientos técnicos de desarrollo software pero tiene claro el resultado final.

Los casos de uso se desarrollan en forma de tabla, y utilizando un lenguaje lo más natural, menos técnico y más cercano al usuario final posible.

La razón de su importancia es que serán de gran utilidad tanto para el usuario final (porque podrá exponer de forma no técnica, pero concisa, sus necesidades) como a los desarrolladores (que utilizarán estos casos de uso para describir los requisitos funcionales de la aplicación).

A continuación se muestran los casos de uso que se han detallado para las necesidades del sistema a desarrollar.

Puesto que el proyecto requiere del desarrollo de dos sistemas (Cliente y Servidor), se distinguen dos tipos de casos de uso: Casos de Uso de Cliente, y Casos de Uso del Servidor.

3.1.1. Casos de Uso del Cliente

Nombre	Registrarse en el sistema
Autor	Antonio Laguna
Fecha	5-9-2013
Descripción:	
Permite al nuevo usuario del sistema darse de alta en el servicio	
Actores:	
Usuario, Servidor	
Precondiciones:	
El usuario debe haber instalado la aplicación	
Flujo Normal:	
1. El usuario rellena un formulario, indicando su número de teléfono y eligiendo una contraseña para acceder al sistema. 2. El usuario pulsa sobre el botón enviar. 3. El sistema envía una petición al servidor con los datos del formulario	
Flujo Alternativo:	
1. El usuario rellena un formulario, indicando su número de teléfono y eligiendo una contraseña para acceder al sistema. 2. El usuario pulsa sobre el botón enviar. 3. El sistema no envía la petición al servidor porque los campos “contraseña” y “confirmar contraseña” son iguales 4. Se muestra un mensaje de error al usuario y se le insta a que revise los campos y vuelva a intentarlo.	
Postcondiciones:	
El cliente se loguea en el servidor con los datos del registro.	

Nombre	Loguearse en el sistema
Autor	Antonio Laguna
Fecha	5-9-2013
Descripción:	
Permite al usuario acceder al sistema, demostrando al servidor su identidad	
Actores:	
Usuario, Servidor	
Precondiciones:	
El usuario debe estar registrado en el sistema	
Flujo Normal:	
<ol style="list-style-type: none"> 1. El usuario rellena un formulario indicando su número de teléfono y su contraseña, elegida a la hora de registrarse 2. El Usuario pulsa sobre el botón enviar. 3. El sistema envía una petición de login al servidor con los datos del formulario. 4. El sistema permite acceder al usuario al menú de conversaciones 	
Flujo Alternativo:	
<ol style="list-style-type: none"> 1. El usuario rellena un formulario indicando su número de teléfono y su contraseña, elegida a la hora de registrarse 2. El Usuario pulsa sobre el botón enviar. 3. El sistema envía una petición de login al servidor con los datos del formulario. 4. Se muestra un mensaje de error al usuario y se le insta a que revise los campos y vuelva a intentarlo. 	
Postcondiciones:	
El usuario deberá revisar los datos introducidos y volver a enviar una petición de login al servidor.	

Nombre	Abrir nueva conversación
Autor	Antonio Laguna
Fecha	5-9-2013
Descripción:	
El usuario abre una nueva conversación con otro usuario con el fin de enviarle mensajes	
Actores:	
Usuario	
Precondiciones:	
El usuario debe estar logueado en el sistema	
Flujo Normal:	
<ol style="list-style-type: none"> 1. El usuario abre el menú de contactos pulsando sobre el botón correspondiente. 2. El usuario selecciona un contacto de la lista de sus contactos que se le muestra por pantalla. 	
Flujo Alternativo:	
Postcondiciones:	
La conversación, aún vacía, queda almacenada en menú principal del usuario, con el fin de acceder rápidamente a ella.	

Nombre	Enviar mensaje
Autor	Antonio Laguna
Fecha	5-9-2013
Descripción:	
Permite al usuario enviar un mensaje de texto a otro usuario de la aplicación	
Actores:	
Usuario, Servidor	
Precondiciones:	
<ul style="list-style-type: none"> - El usuario emisor debe estar logueado - El usuario emisor debe tener abierta la conversación con el usuario receptor 	
Flujo Normal:	
<ol style="list-style-type: none"> 1. El usuario abre una conversación con el usuario receptor 2. El usuario escribe su mensaje en el campo inferior. 3. El usuario pulsa el botón enviar. 4. El sistema crea el mensaje y lo envía al servidor. 5. El sistema nos muestra el mensaje en la conversación como enviado. 	
Flujo Alternativo:	
<ol style="list-style-type: none"> 1. El usuario abre una conversación con el usuario receptor 2. El usuario escribe su mensaje en el campo inferior. 3. El usuario pulsa el botón enviar. 4. El sistema crea el mensaje y lo envía al servidor. 5. El sistema almacena el mensaje como pendiente de envío, ya que ha sido imposible enviarlo por falta de conexión. 	
Postcondiciones:	
-El sistema almacena el mensaje en la base de datos local	

Nombre	Crear Grupo de Conversación
Autor	Antonio Laguna
Fecha	5-9-2013
Descripción:	
Permite al usuario crear un nuevo grupo de conversación, seleccionando varios de sus contactos	
Actores:	
Usuario, Servidor	
Precondiciones:	
<ul style="list-style-type: none"> - El usuario emisor debe estar logueado 	
Flujo Normal:	
<ol style="list-style-type: none"> 1. El usuario, pulsa el botón de “nueva conversación” en su menú principal. 2. El usuario selecciona los contactos que desea incluir en sus grupo de conversación de una lista, y elige el nombre para el grupo 3. El usuario pulsa el botón de “crear grupo” 4. El sistema envía una petición de creación de grupo al servidor con los contactos seleccionados. 5. El usuario emisor, y los que has seleccionado reciben la notificación de que un nuevo grupo ha sido creado. 	
Flujo Alternativo:	
<ol style="list-style-type: none"> 1. El usuario, pulsa el botón de “nueva conversación” en su menú principal. 2. El usuario no selecciona ningún contacto y pulsa el botón crear 3. El sistema le notificará de que debe seleccionar al menos dos contactos para crear un grupo y no le permitirá crearlo sin contactos seleccionados. 	
Postcondiciones:	
El grupo creado quedará almacenado como una conversación en el menú principal	

Nombre	Cerrar la aplicación
Autor	Antonio Laguna
Fecha	5-9-2013
Descripción:	
Permite al usuario salir del programa, desloguearse y no recibir mensajes entrantes hasta que se vuelva a iniciar	
Actores:	
Usuario	
Precondiciones:	
Estar logueado	
Flujo Normal:	
<ol style="list-style-type: none"> 1. El usuario pulsa el botón salir del menú principal 2. El sistema envía una petición de deslogueo al servidor 3. El sistema se cierra de forma completa 	
Flujo Alternativo:	
Postcondiciones:	
<ul style="list-style-type: none"> -El sistema se cierra de forma completa -No existen conexiones ni procesos trabajando con el servidor 	

3.1.2. Casos de uso del Servidor

Nombre	Recibir petición de registro
Autor	Antonio Laguna
Fecha	5-9-2013
Descripción:	
El Servidor recibe una petición de registro de un usuario de la aplicación	
Actores:	
Usuario, Servidor	
Precondiciones:	
Flujo Normal:	
1. Al servidor le llega una petición de registro, con el par : numero, contraseña 2. Se comprueba que ese número no esté registrado 3. El servidor almacena el par “número-contraseña” en la base de datos 4. El servidor envía un mensaje confirmando el registro del usuario con esos datos.	
Flujo Alternativo:	
1. Al servidor le llega una petición de registro, con el par : numero, contraseña 2. Se comprueba que ese número no esté registrado 3. El servidor devuelve al usuario un mensaje de error debido a que ese número ya está registrado en el sistema	
Postcondiciones:	

Nombre	Recibir petición de login
Autor	Antonio Laguna
Fecha	5-9-2013
Descripción:	
El Servidor recibe una petición de login de un usuario de la aplicación	
Actores:	
Usuario, Servidor	
Precondiciones:	
Flujo Normal:	
<ol style="list-style-type: none"> 1. Al servidor le llega una petición de login, con el par : numero, contraseña 2. Se comprueba que ese número esté registrado 3. Se comprueba que en la base de datos exista el par “numero-contraseña” recibido 4. El servidor envía un mensaje confirmando la correcta identificación del usuario en el sistema 	
Flujo Alternativo:	
<ol style="list-style-type: none"> 1. Al servidor le llega una petición de login, con el par : numero, contraseña 2. Se comprueba que ese número esté registrado 3. Se comprueba que en la base de datos exista el par “numero-contraseña” recibido 4. El servidor envía un mensaje de error al usuario debido a que no existe un par “número-contraseña” como el que ha enviado. 	
Postcondiciones:	

Nombre	Recibir mensaje
Autor	Antonio Laguna
Fecha	5-9-2013
Descripción:	
El Servidor recibe un mensaje de un usuario de la aplicación con destino otro usuario de la misma	
Actores:	
Usuario, Servidor	
Precondiciones:	
Usuario registrado y logueado en el sistema	
Flujo Normal:	
<ol style="list-style-type: none"> 1. Al servidor le llega un mensaje para un usuario de la aplicación 2. Se comprueba que el usuario receptor esté registrado en el sistema. 3. El Servidor almacena el mensaje en su base de datos, a la espera de que el receptor pida el mensaje. 	
Flujo Alternativo:	
<ol style="list-style-type: none"> 1. Al servidor le llega un mensaje para un usuario de la aplicación 2. Se comprueba que el usuario receptor esté registrado en el sistema. 3. El usuario receptor no está registrado en el sistema, por lo que el mensaje se elimina. 	
Postcondiciones:	

Nombre	Petición de mensajes
Autor	Antonio Laguna
Fecha	5-9-2013
Descripción:	
El servidor recibe una petición de un usuario para que le envíe sus mensajes.	
Actores:	
Usuario, Servidor	
Precondiciones:	
Usuario registrado y logueado en el sistema	
Flujo Normal:	
<ol style="list-style-type: none"> 1. Al servidor le llega una petición para enviarle los mensajes a un cliente 2. Se comprueba que el usuario receptor esté registrado en el sistema. 3. Se buscan sus mensajes en la base de datos del servidor. 4. El servidor envía los mensajes al usuario como respuesta a sus petición. 	
Flujo Alternativo:	
<ol style="list-style-type: none"> 1. Al servidor le llega una petición para enviarle los mensajes a un cliente 2. Se comprueba que el usuario receptor esté registrado en el sistema. 3. Se buscan sus mensajes en la base de datos del servidor. 4. El servidor no envía ningún mensaje al cliente porque no tiene mensajes pendientes. 	
Postcondiciones:	
Los mensajes enviados a los clientes deben ser borrados de la base de datos del servidor.	

3.1.3. Diagrama de casos de uso

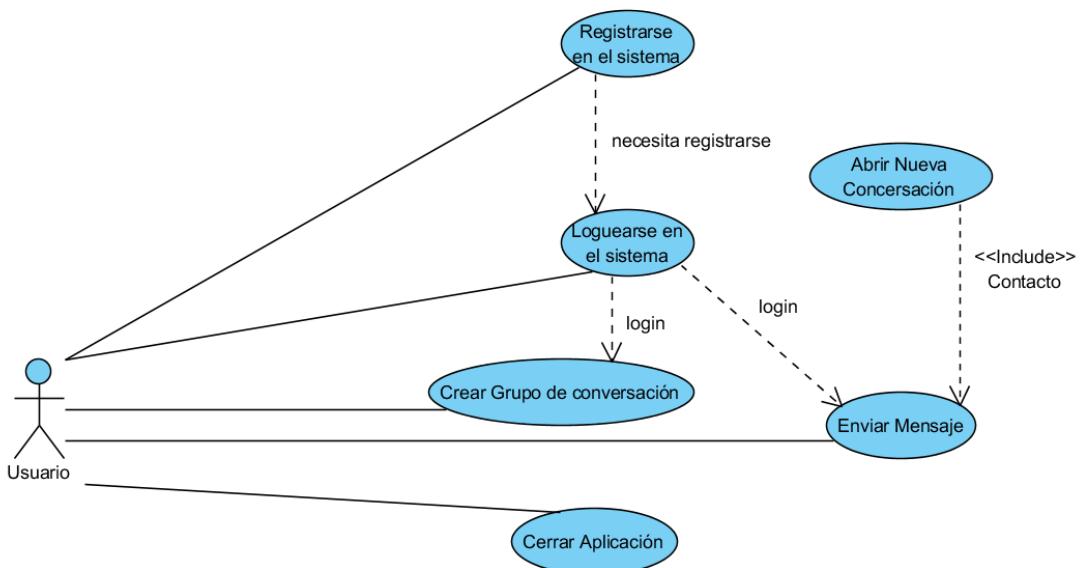


Figura 21: Diagrama de casos de uso del cliente

3.2. Análisis de requisitos

El análisis de requisitos es un proceso que abarca todas las actividades relacionadas con la determinación de los objetivos, necesidades y funciones del sistema a desarrollar.

Es un proceso largo y muy detallado. El estándar IEEE 830-1998 [28] abarca de forma profunda todos estos aspectos. Sin embargo el desarrollo del análisis del sistema basándonos estrictamente en este estándar sería un proceso demasiado exhaustivo, y posiblemente no encaje con nuestras necesidades y sea algo inabordable. Por tanto se ha tomado la decisión de basarnos en el estándar IEEE 830-1998 pero no de forma estricta, sino adaptándolo a nuestras necesidades. De esta forma no desarrollaremos en profundidad el ámbito, diccionario de datos y distintos agentes, sino que nos limitaremos a una correcta definición de los requisitos, tanto funcionales como de sistema, y la relación existente entre ellos.

Como se ha comentado en secciones anteriores, el presente TFG consta de dos partes bien diferenciadas, el servidor, y el cliente, así que se analizarán los requisitos para ambos sistemas.

3.2.1. Requisitos Funcionales

Los requisitos funcionales implementan los casos de uso, y definen la capacidad de un sistema. En ocasiones también definen lo que el sistema no debe hacer.

3.2.1.1. Requisitos Funcionales Servidor

[RF S1] El Servidor recibirá los mensajes que todos los usuarios envíen a otros usuarios, y los almacenará en su base de datos

[RF S2] El Servidor enviará mensajes a los usuarios que previamente ha recibido de otros usuarios.

[RF S3] El Servidor debe eliminar de su base de datos cada mensaje que ha sido recibido por el receptor.

[RF S4] El servidor debe ser capaz de registrar nuevos usuarios en el sistema, a petición de los usuarios.

3.2.1.2. Requisitos Funcionales Cliente

[RF C1] Se debe mostrar al usuario una pantalla de acceso / registro la primera vez que se inicia el programa. Una vez que se haya accedido al sistema correctamente, el programa almacenará el usuario y la contraseña en el teléfono de manera que el acceso en posteriores ocasiones sea de forma automática.

[RF C2] El usuario podrá ver una lista de sus contactos, y seleccionar aquel con el que quiere comenzar una conversación.

[RF C3] El usuario puede acceder a una lista de sus conversaciones.

[RF C4] El sistema almacenará los mensajes de forma persistente

[RF C5] El usuario podrá cerrar de forma completa la aplicación, y no recibir mensajes a través de esta hasta que vuelva a ser abierta.

[RF C6] El usuario podrá iniciar una conversación de grupo, en la que podrá elegir qué usuarios de su agenda participarán en ella.

[RF C7] El usuario podrá enviar un mensaje a un contacto

[RF C8] El usuario podrá recibir mensajes de otros usuarios

3.2.2. Requisitos no funcionales

Los requisitos no funcionales son requisitos del sistema que no afectan directamente al usuario, sino al modo de hacer las cosas. Digamos que los requisitos funcionales especifican qué debe hacer el sistema, y los requisitos no funcionales especifican cómo debe hacerlo.

Dentro del grupo de requisitos no funcionales, se podrían distinguir distintos grupos de requisitos, como requisitos de seguridad, de comunicaciones o de rendimiento. No obstante en el presente TFG se abarcarán todos estos requisitos de forma general, como requisitos no funcionales.

3.2.2.1. Requisitos No Funcionales Servidor

[RNF S1] El servidor debe estar conectado a una base de datos donde almacene toda la información relevante.

[RNF S2] El servidor velará por que la identidad de los usuarios sea confirmada, evitando suplantaciones de identidad

[RNF S3] El servidor velará por la seguridad de los datos de sus usuarios

3.2.2.2. Requisitos No Funcionales Cliente

[RNF C1] El cliente enviará los mensajes en formato JSON, muy útil para la transmisión de información.

[RNF C2] El cliente capturará los mensajes entrantes en segundo plano, de forma que la interfaz de usuario nunca sea bloqueada.

[RNF C3] El cliente estará en ejecución siempre, hasta que el usuario pulse sobre el botón salir. El sistema Android no cerrará automáticamente el programa.

[RNF C4] El usuario no puede saber si el receptor ha recibido o no el mensaje enviado, tan solo tendrá constancia de que el servidor lo ha recibido y almacenado correctamente.

[RNF C6] El usuario no podrá saber cuándo fue la última vez en la que sus contactos se conectaron al sistema.

4. Desarrollo y etapas

Como metodología de trabajo se utilizará un modelo en espiral.

El desarrollo en espiral, definido por primera vez por Barry Boehm[29], es un modelo de ciclo de vida software considerado rápido y eficiente. Es adecuado para proyectos en los que se tienen claros los objetivos finales, pero no todos los detalles de implementación.

En este proyecto se ha utilizado una tecnología con la que el alumno no tenía experiencia. Para ir perfilando los requisitos y algunos detalles de implementación se han realizado muchas reuniones con el director del T.F.G., en las que se han ido elaborando bocetos del diseño de los dos sistemas que conforman el Software. Estos bocetos se muestran en las siguientes páginas.

El proyecto se divide en módulos más pequeños y a estos módulos se le aplican 4 fases:

- **Determinar Objetivos:** En esta fase se determina el alcance del módulo, alternativas, requisitos y restricciones.
- **Análisis de riesgos:** En esta fase se analizan detalladamente cada uno de los riesgos del sistema y se definen pasos a seguir para reducirlos.
- **Desarrollar, Verificar y Validar:** En esta fase se implementa el módulo correspondiente, quedando operativo y verificando que cumpla los objetivos determinados. Esta fase incluye la validación, una serie de pruebas que asegurarán el correcto funcionamiento del módulo.
- **Planificación:** En esta fase, se revisa la parte hecha del proyecto, evaluando si es necesario otro ciclo más de espiral.

Cada iteración proporcionará un prototipo funcional, y la iteración final nos proporcionará el sistema completo.

La siguiente ilustración (Figura 22) explica muy claramente el proceso de desarrollo en espiral, tomada de [38].

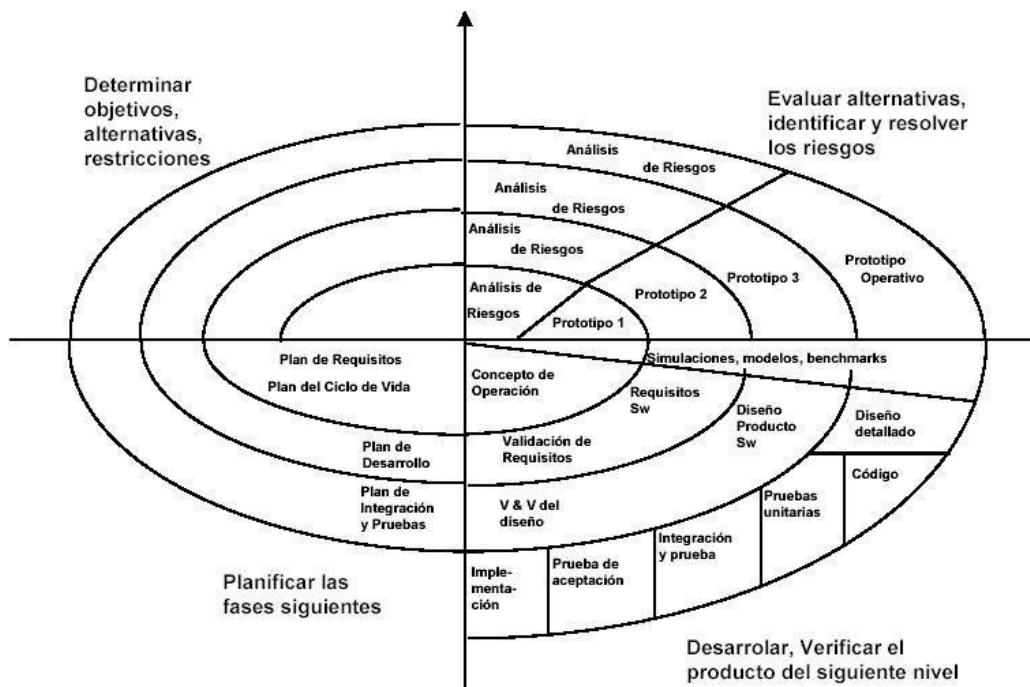


Figura 22: Ejemplo de desarrollo en espiral

Como ya se ha mencionado, el proyecto consta de un sistema cliente y un sistema servidor. Para el correcto desarrollo del sistema, se ha optado por desarrollar ambos sistemas paralelamente, de forma que al final de cada iteración siempre habrá un prototipo funcional tanto del cliente como del servidor.

4.1. Riesgos iniciales y cuestiones preliminares

Es importante destacar que, al comenzar este TFG, su autor carecía por completo de experiencia en el desarrollo de aplicaciones para dispositivos móviles. Existía, por tanto, un riesgo elevado de “atascarse” y retrasar de manera importante el avance del trabajo.

Por ello se decidió abordar una primera iteración que sirviese como prueba de concepto y que permitiese al alumno familiarizarse con el entorno tecnológico.

Si esta prueba se superaba de forma satisfactoria, entonces se podría continuar con el desarrollo del sistema.

Otros riesgos previsibles están relacionados con las características del dispositivo cliente: pantalla, widgets, consumo de batería, etc.

4.2. Ciclos

A continuación, se detallará el proceso llevado a cabo para la elaboración del sistema descrito.

Como hemos explicado anteriormente, la metodología de desarrollo utilizada ha sido el modelo en espiral o prototipado, así que a continuación se describirán las diferentes iteraciones y prototipos del sistema hasta la finalización de este.

4.2.1. Ciclo 1: Prueba de Concepto. Envío de mensajes

Determinar Objetivos

El objetivo de este primer prototipo es el desarrollo de un pequeño servidor en JSP y un pequeño cliente en Java (Android). Este sistema gestionará el envío de mensajes de texto desde varios clientes, diferenciados por su IP, hasta el servidor, y este mostrará los mensajes por pantalla (Figura 23).

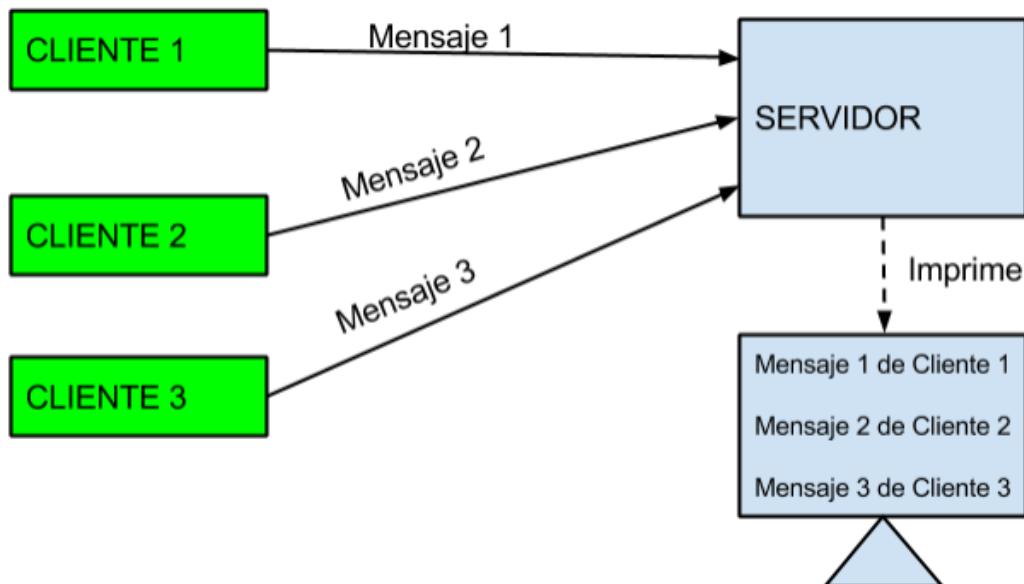


Figura 23: Objetivo ciclo 1

Análisis de riesgos

En este primer prototipo se va a desarrollar una prueba de concepto, que comunicará el cliente con el Servidor, permitiendo al cliente enviar mensajes al Servidor, para comprobar que los ha recibido los imprimirá por pantalla, indicando su origen.

Podemos considerar esta primera iteración como una toma de contacto con las tecnologías que nos permitirá decidirnos por continuar o por abandonar el proyecto.

Desarrollar, Verificar y Validar

Se utilizará un modelo de desarrollo en 4 capas (interfaz, dominio, persistencia y comunicaciones) tanto en el cliente como en el servidor. Esto permitirá que la aplicación sea mucho más clara y modificable. En caso de no hacerlo así podríamos encontrarnos frente a un sistema rígido, inestable e imposible de mantener. Se ha pensado una posible solución para los requisitos de este primer ciclo, y queda plasmado en los siguientes diagramas UML de clases, que se elaboraban a mano en las primeras reuniones con el director del TFG (Figuras 24 y 25)

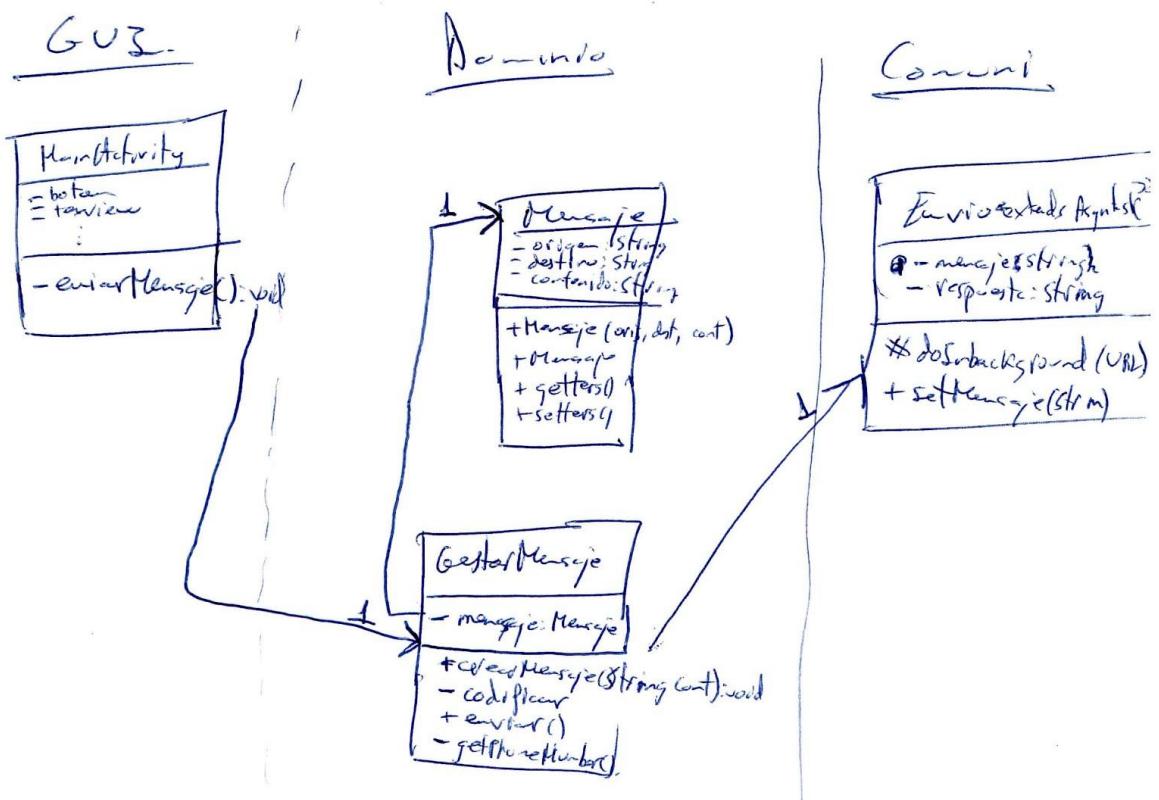


Figura 24: Diagrama de clases del Cliente (ciclo 1)

Como se aprecia en la Figura 24, el cliente tiene una sola interfaz de usuario, desde la que construye objetos del tipo mensaje, apoyándose en la clase GestorMensaje. Una vez construidos los mensajes, se envían al servidor utilizando la clase Envío.

Se puede destacar que la clase Envío extiende de Asyntask, ya que es un requisito de Android que las operaciones de red se realicen en segundo plano.

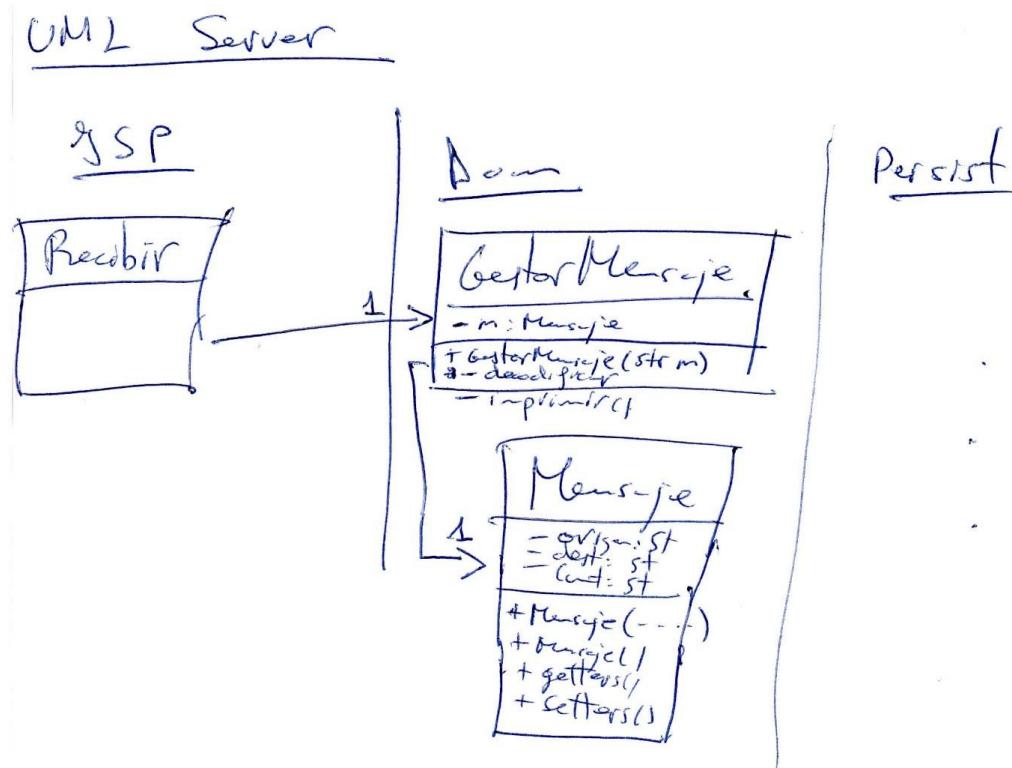


Figura 25: Diagrama de clases del Servidor (ciclo 1)

En cuanto al servidor, consta de momento de una sola página JSP, que recibe los mensajes enviados por el cliente y los gestiona apoyándose en la clase GestorMensaje. Como se aprecia, aún no se ha implementado la persistencia de datos, por los que los mensajes aún no se almacenan, sino que únicamente pueden mostrarse por pantalla.

Resulta interesante incluir el diagrama original, que fue escrito en papel, en lugar de hacerlo en una herramienta de modelado como Visual Paradigm ya que de momento la complejidad del diagrama es pequeña y puede ser descrito sin apoyarnos en herramientas externas.

La interfaz del cliente tendrá un solo textbox y un botón. Pulsando el botón, el cliente enviará el contenido del textbox al servidor (Figura 26).

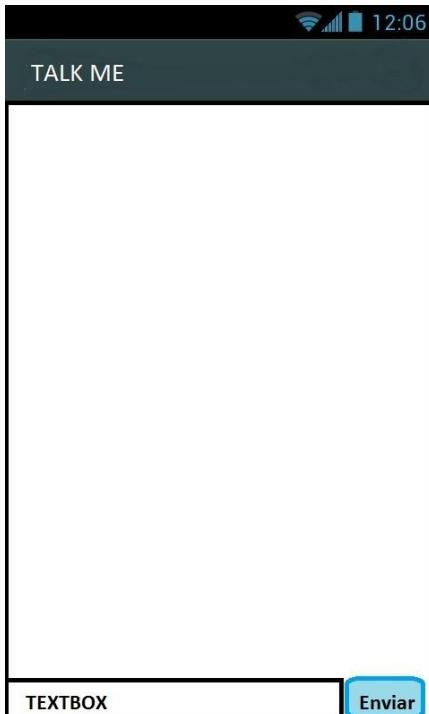


Figura 26: Interfaz Cliente (ciclo 1)

Para cada envío de mensaje, se ha tomado la decisión de crear un hilo de ejecución distinto, ya que el envío es un proceso lento, y podría bloquear nuestra interfaz gráfica. De hecho, el propio Android nos impide realizar una operación de red desde el hilo principal. Para evitar esto, ejecutamos cada envío en un hilo secundario.

```
100
101         botonEnviar.setOnClickListener(new OnClickListener() {
102             @Override
103             public void onClick(View v) {
104                 if(!cuadroMensaje.getText().toString().equals ""){
105                     String mensaje=cuadroMensaje.getText().toString();
106                     cuadroMensaje.setText("");
107                     enviarMensaje(mensaje);
108                 }
109             }
110
111
112         });
113     }
114 );
115 }
116 }
117
118 private void enviarMensaje(String mensaje){
119     final GestorMensaje gm=new GestorMensaje(mensaje,miNumero,numero);
120     new Thread(new Runnable() {
121         public void run() {
122             gm.enviar(getApplicationContext(),1,1);
123         }
124     }).start();
125 }
126 }
```

Figura 27: Enviar Mensaje. GUIConversacion.java

Antes de enviar el mensaje (Figura 27), el cliente transformará el objeto Mensaje a un formato de intercambio de información, JSON, con el fin de que el servidor pueda interpretarlo correctamente (Figura 28). JSON, es un estándar, y existen muchas implementaciones de él. La más interesante nos ha parecido GSON[29], de Google

```

private String codificar(){
    Gson g=new Gson();
    String mensajeEnJson=g.toJson(mensaje);
    return mensajeEnJson;
}

public boolean enviar(Context contexto,int intento,int tipo){
    boolean enviado=false;
    String mensajeCodificado=codificar();
    long sumaComprobacion=calcularCRC32(mensajeCodificado);
    EnvioMensajeHilo env=new EnvioMensajeHilo();
    URL direccion = null;
    try {
        if(+>+>-1){
}
}

```

Figura 28: codificar. GestorMensaje.java

Para el envío de mensajes al servidor, utilizamos el protocolo HTTP. Para ello, optamos utilizar las librerías de Apache HTTP [31].

```

33
34     public void enviar(URL params, String tipo) {
35         String url = params.toString();
36         HttpClient client = new DefaultHttpClient();
37         HttpPost post = new HttpPost(url);
38         List<NameValuePair> urlParameters = new ArrayList<NameValuePair>();
39         urlParameters.add(new BasicNameValuePair(tipo, this.mensaje));
40
41         try {
42             post.setEntity(new UrlEncodedFormEntity(urlParameters));
43
44         } catch (UnsupportedEncodingException e) {
45             e.printStackTrace();
46         }
47         try {
48
49             HttpResponse respuesta=client.execute(post);
50
51             HttpEntity entity = respuesta.getEntity();
52             InputStream is = entity.getContent();
53
54             this.respuesta= limpiarRespuesta(is);
55
56
57         } catch (ClientProtocolException e) {
58             e.printStackTrace();
59         } catch (IOException e) {
60             e.printStackTrace();
61         } catch(Exception e){
62             e.printStackTrace();
63         }
64     }
}

```

Figura 29: enviar. Envio.java

Como se aprecia en la figura anterior (Figura 29); **Error! No se encuentra el origen de la referencia.** se crea un cliente HTTP y se le introducen como parámetros url el mensaje previamente codificado en JSON (Figura 28). Una vez preparado, se ejecuta el cliente y se espera respuesta.

El servidor recibe el mensaje de la siguiente forma:

```

13  String texto=null;
14  try{
15      texto=request.getParameter("mensaje");
16      System.out.println(request.getRemoteHost());
17      GestorMensaje gm=new GestorMensaje(texto);
18      gm.imprimirPorPantalla();
19      Long suma = gm.getSuma();
20      %>
21      <%=suma%>
22  <%
23 }catch(Exception e){
```

Figura 30: recibir.jsp

El proceso de recepción del mensaje es sencillo, como expone la figura anterior (Figura 30) basta con utilizar la función getParameter() para acceder a los parámetros URL de la petición HTTP recibida.

Planificación

Requisitos del cliente:

Requisito	Pendiente	En Proceso	Completo
RF C1	X		
RF C2	X		
RF C3	X		
RF C4	X		
RF C5	X		
RF C6	X		
RF C7		X	
RF C8	X		

Requisitos del servidor

Requisito	Pendiente	En Proceso	Completo
RF S1		X	
RF S2	X		
RF S3	X		
RF S4	X		

El sistema actual permite enviar mensajes desde el cliente al servidor, y este los imprime por pantalla.

De momento no hay persistencia de datos, ni en el servidor, ni en el cliente. Se implementarán en posteriores ciclos de desarrollo.

El cliente accede directamente a la interfaz de mensajes de archivos “GUIConversacion”. De momento la ip del servidor está en el propio código, y el destino del mensaje no se puede modificar de momento.

No hay posibilidad de registrarse en el sistema, ni de loguearse, aunque en esta fase tampoco es necesario, se implementará más adelante.

En la siguiente iteración, se prevé implementar un sistema de login y registro para el usuario a través del cual podrá acceder a la aplicación.

4.2.2. Ciclo 2: Instalación y registro. Persistencia en el Servidor

Determinar Objetivos

En esta fase la idea es que no todo el mundo pueda acceder al sistema del ciclo 1, sino que sea necesario registrarte y posteriormente loguearse en el sistema. Para ello necesitaremos implementar los siguientes mecanismos:

- Persistencia en el servidor: Necesitaremos implementar una base de datos en el servidor para tener constancia persistente de los usuarios que se encuentran registrados en el sistema
- Sistema de registro en el cliente: Un formulario que nos pida información para registrarnos en el sistema, y envíe una petición de registro al servidor.
- Sistema de login en el cliente: Un formulario que nos permita introducir nuestros datos de acceso al sistema, y los envíe al servidor para comprobar su veracidad.
- Instalación: Consiste en un sistema para almacenar los datos de acceso al sistema de modo que una vez logueados la primera vez, las posteriores se hagan de forma automática, sin necesidad de introducir los datos manualmente.
- Bienvenida: Será la primera interfaz con la que se encontrará el usuario del sistema. Nos permitirá elegir entre loguearnos en el sistema o darnos de alta (registro). Una vez nos hayamos logueado en el sistema, esta pantalla nunca más aparecerá, porque se accederá a los datos almacenados de logueo.

Análisis de riesgos

R1: Problemas con la gestión de persistencia

Probabilidad: muy baja, ya que se dispone de experiencia suficiente.

Desarrollar, Verificar y Validar

Para la persistencia del servidor se ha elegido el motor de bases de datos mySQL[32] por su gran potencia y compatibilidad. Para la creación y gestión de la base de datos se ha utilizado la herramienta phpMyAdmin[33].

Para almacenar los datos de acceso al sistema, se ha optado por no hacerlo en una bbdd relacional sino en un fichero de texto plano “instalacion.cnf”

A partir de los requisitos de este ciclo, se diseñó el siguiente diagrama de clases en una reunión con el director para planificar el prototipo (Figura 31 y Figura 32)

Al abrir la aplicación se ejecuta un activity que comprueba si es la primera vez que intentamos acceder al sistema. En caso afirmativo, se nos muestra un menú donde se nos dará a elegir entre crear un nuevo usuario o identificarnos con un usuario existente. Sea cual sea nuestra respuesta, se nos abrirá otro activity para introducir nuestros datos e identificarnos, y en caso de que hayamos elegido esta opción, o registrarnos en caso de haber elegido la otra. Cuando nos identifiquemos en el sistema de forma satisfactoria, la aplicación almacenará nuestro usuario y contraseña en un fichero, para que la próxima vez que abramos la aplicación nos saltemos este proceso y accedamos directamente al sistema.

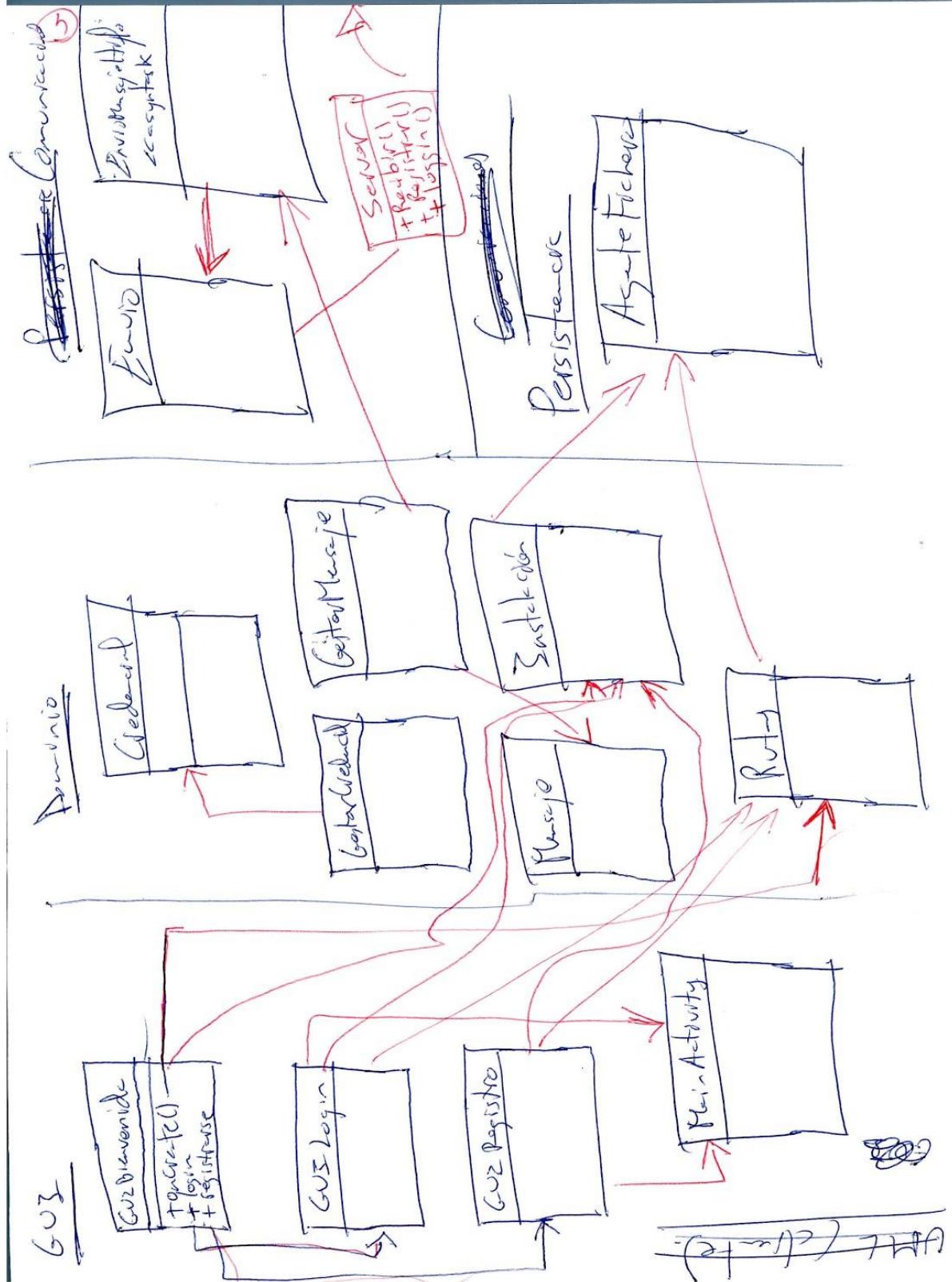


Figura 31: Diagrama de clases Cliente. Ciclo 2

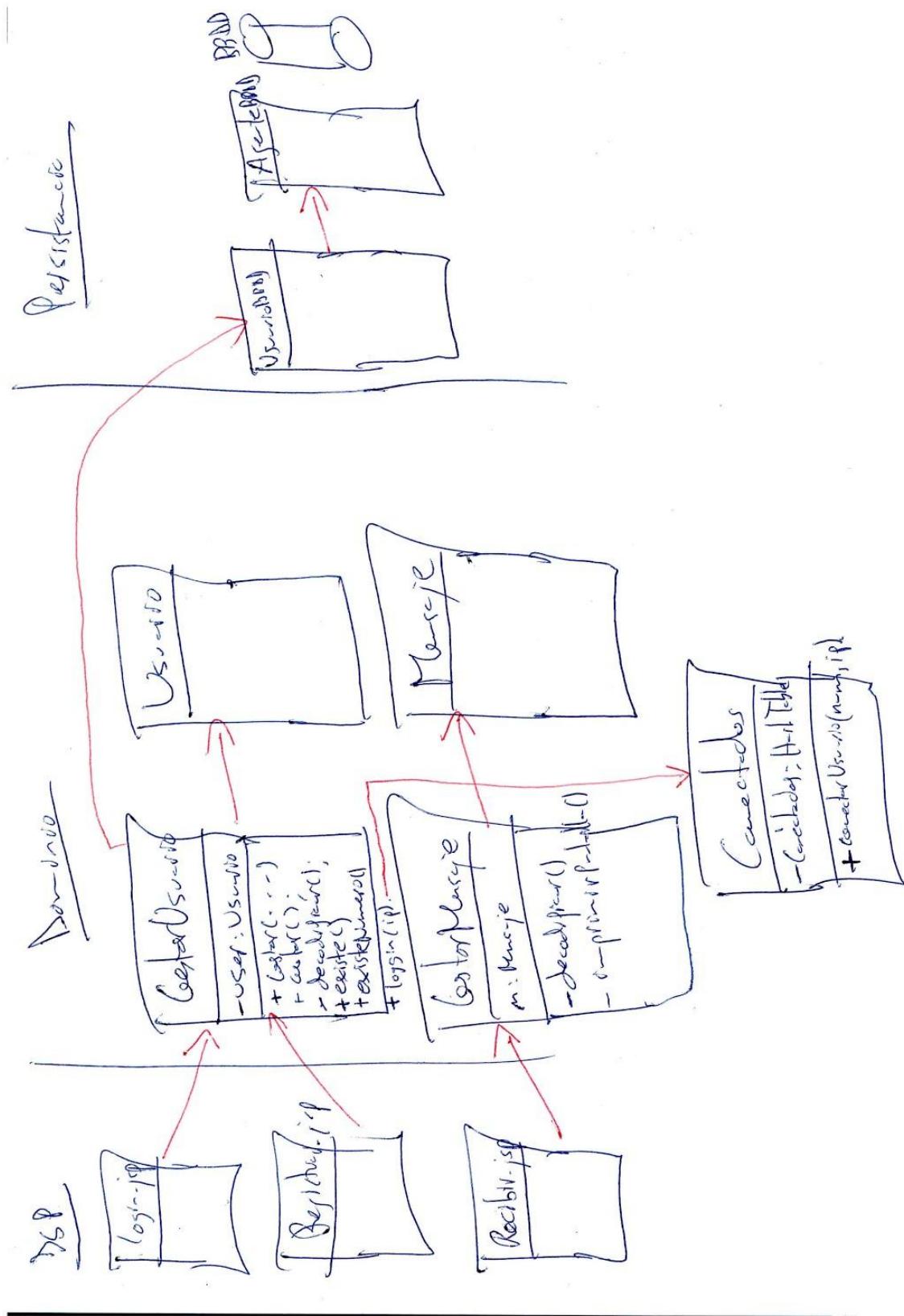


Figura 32: Diagrama de clases Servidor. Ciclo 2

En este ciclo, la experiencia de usuario será la siguiente:

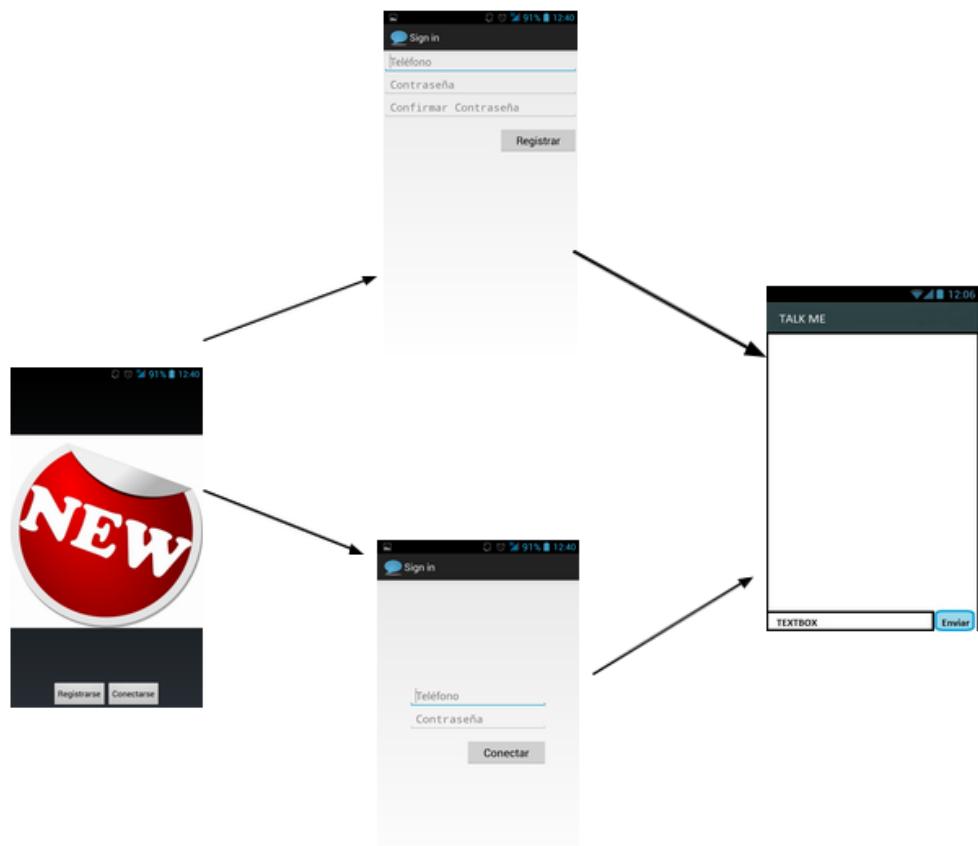


Figura 33: Experiencia de usuario. Ciclo 2

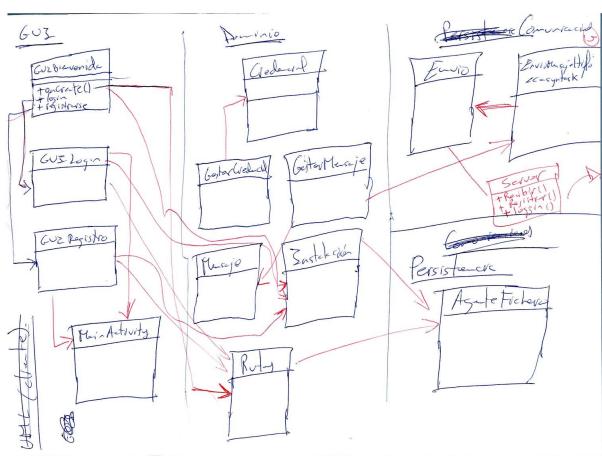


Figura 34: Reproducción de la Figura 30

La primera interfaz, donde elegimos identificarnos o registrarnos, es una activity llamado GUIBienvenida en **¡Error! No se encuentra el origen de la referencia..** La interfaz de registro es GUIRegistro, y la de identificación es GUILogin. La interfaz de mensajes donde accedemos al identificarnos aún mantiene un nombre genérico llamado MainActivity en el diagrama de clases de la figura anterior.

Es un ciclo bastante extenso ya que implementa varias funciones muy importantes, así que se intentarán detallar lo máximo posible.

En primer lugar se detallará la implementación del registro y logueo de usuarios del sistema en el servidor, y cómo implementa su base de datos. Después se analizará cómo implementar estas funciones en el cliente.

Para implementar la gestión de usuarios en el servidor, lo primero es crear una base de datos, y dentro de esta una tabla usuario (numero, pass) (**¡Error! No se encuentra el origen de la referencia.**

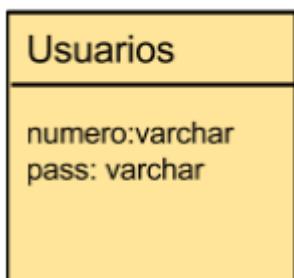


Figura 35: BBDD Servidor. Ciclo 2

La base de datos la llamamos “servidorbbdd”. Como nos encontramos en las primeras iteraciones utilizamos el usuario ya creado “root” y la contraseña vacía.

```
20     private static void conectar() throws Exception {
21         //bdd local
22         String url="jdbc:mysql://localhost:3306/servidorbbdd";
23         //bdd internet
24         //String url="jdbc:mysql://db4free.net:3306/bbddchat";
25         String driver="com.mysql.jdbc.Driver";
26         Class.forName(driver);
27         mBD=DriverManager.getConnection(url, "root", "");
28     }
29 }
```

Figura 36: Conectar. Agente.java

Como puede apreciarse en Figura 36 (línea 24), se barajó la posibilidad de externalizar la base de datos, utilizando un servidor gratuito en internet, pero esta opción se descartó tras las primeras pruebas debido a que era demasiado lenta comparándola con la base de

datos instalada en el propio servidor. Así pues nos decantamos por una base de datos en el mismo servidor.

Una vez recibida una petición de registro el proceso será el siguiente:

1. Recibimos una petición de registro, con un argumento de tipo Usuario, con atributos número y pass, y una IP que podemos obtener de la cabecera HTTP del mensaje entrante.
2. Decodificamos el objeto Usuario que está codificado en formato JSON.
3. Comprobamos que NO existe el usuario en la tabla Usuarios de la base de datos.
4. Si no existe, insertamos un nuevo registro en la tabla usuarios con los datos recibidos del cliente. Si existe, se enviará un mensaje de error al cliente.

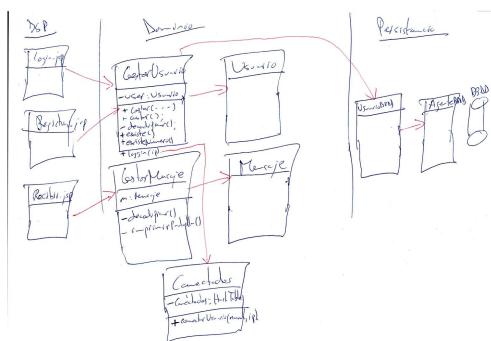


Figura 37: Reproducción de la Figura 31

En la figura 38 se aprecia como el servidor a través del método getParameter() descarga el contenido del mensaje enviado por el cliente. Este mensaje contiene los datos del usuario codificados en JSON, que el gestor Usuario se encargará de descodificarlo y almacenarlos en la base de datos (Figura 39 y Figura 40), siempre que cumplan las condiciones de no estar ya registrado y se pueda descodificar correctamente.

```
11  <%
12 |String mensaje=request.getParameter("mensaje");
13 GestorUsuario gu=new GestorUsuario(mensaje);
14 ▼ if(gu.existeNumero()){
15     %>fail<%
16     System.out.println("fallo al registrar");
17 ▼ }else{
18     gu.guardar();
19     System.out.println("registrado usuario correctamente");
20     %>ok<%
21 }
22
```

Figura 38:Login.jsp

```
53     public boolean existeNumero(){
54         UsuarioBBDD ubbdd=new UsuarioBBDD(this.user);
55         return ubbdd.existeNumEnBaseDeDatos();
56     }
57
58     public void guardar(){
59         if(this.user!=null){
60             UsuarioBBDD gbd=new UsuarioBBDD(this.user);
61             gbd.insertarEnBaseDeDatos();
62             System.out.println("guardado");
63         }
64     }
65
```

Figura 39: guardar. GestorUsuario.java

```
42
43     public void insertarEnBaseDeDatos(){
44         String telefono=this.usuario.getNumero();
45         String pass=this.usuario.getContraseña();
46         try {
47             Agente.getAgente().insert("insert into usuarios values('"+telefono+"','"+pass+"')");
48         } catch (SQLException e) {
49             System.out.println("error al insertar en sql");
50             e.printStackTrace();
51         } catch (Exception e) {
52             // TODO Auto-generated catch block
53             e.printStackTrace();
54         }
55     }
56 }
57
```

Figura 40: insertarEnBaseDeDatos. UsuarioBBDD.java

Con esto, el usuario quedará registrado en el sistema y ya podrá loguearse y acceder al sistema (implementado en el ciclo 1).

Ahora, el sistema podrá recibir peticiones de login del cliente que se ha registrado u otros clientes. El proceso que sigue el servidor para loguear a los usuarios es muy parecido al de las peticiones de registro.

1. Recibimos una petición de login, con un argumento de objeto Usuario, con atributos número y pass, y una IP que podemos obtener de la cabecera HTTP del mensaje entrante. (Figura 41)
2. Decodificamos el objeto Usuario que está codificado en formato JSON.
3. Se comprueba que existe un usuario con ese número de teléfono y esa contraseña en la base de datos. (Figura 42).
4. Si existe, se añade en una tabla hash no persistente, el número de teléfono y la IP del usuario, y enviamos al usuario una respuesta correcta, así podrá entrar en el sistema. Si no existe, se envía un mensaje de error al cliente (Figura 43).

```
11  <%
12  System.out.println("Peticion de login");
13  String texto=request.getParameter("mensaje");
14  GestorUsuario gs=new GestorUsuario(texto);
15  if(gs.existe()){
16      gs.login(request.getRemoteHost());
17      System.out.println("El usuario existe en la bbdd y se ha logueado");
18      %>ok<%
19  }else{
20      System.out.println("Fallo al login. el user no existe en bbdd");
21      %>fail<%
22  }
23
24 %>
```

Figura 41: registrar.jsp

```

58     public boolean existeEnBaseDeDatos(){
59         boolean res=false;
60         Vector vector=new Vector();
61         try {
62             System.out.println(usuario.getNumero());
63             System.out.println(usuario.getContraseña());
64             vector=Agente.getAgente().select("select * from usuarios where(numero='"
65                 +usuario.getNumero()+" "+ "AND pass='"+usuario.getContraseña()+"')");
66         } catch (SQLException e) {
67             // TODO Auto-generated catch block
68             e.printStackTrace();
69         } catch (Exception e) {
70             // TODO Auto-generated catch block
71             e.printStackTrace();
72         }
73         if(vector.size()==1){
74             res=true;
75         }else{
76             res=false;
77         }
78         return res;
79     }

```

Figura 42: existeEnBaseDeDatos. UsuarioBBDD.java

```

66     public boolean loggin(String ip){
67         boolean res=false;
68         Conectados.conectarUsuario(this.user.getNumero(), ip);
69         System.out.println("Usuario: "+this.user.getNumero()+" logueado con ip: "+ip);
70         return res;
71     }
72 }

```

Figura 43: loggin. GestorUsuario.java

Cabe mencionar una vez llegados a este punto, que la clase Agente del servidor, la cual es la encargada del acceso a la base de datos, utiliza un patrón de diseño Singleton, para asegurar que solo exista una única instancia del acceso a la base de datos. (Figura 44)

```

public class Agente {
    protected static Agente mInstancia=null;
    protected static Connection mBD;
    //Constructor
    private Agente()throws Exception {
        Agente.conectar();
    }

    //Implementación del patrón Singleton
    public static Agente getAgente() throws Exception{
        if (mInstancia==null){
            mInstancia=new Agente();
        }
        return mInstancia;
    }
}

```

Figura 44: Patrón Singleton. Agente.java

Por otro lado, el cliente tendrá que implementar una serie de mejoras para alcanzar los objetivos fijados. La siguiente máquina de estados representa el comportamiento de la aplicación que queremos conseguir.

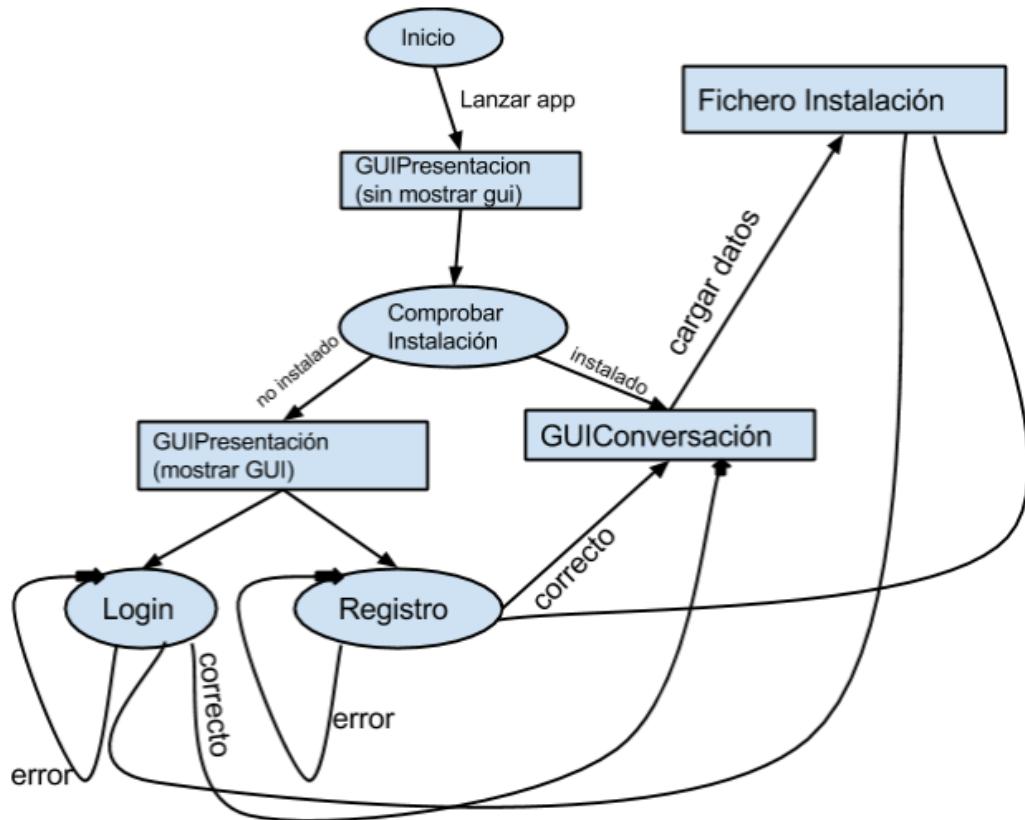


Figura 45: Comportamiento Cliente. Ciclo 2

Nada más arrancar la aplicación, el sistema comprueba si existe el fichero “instalación.cnf”, que es el que almacena los datos de acceso a la aplicación. Si no existe tal fichero significa que nunca hemos entrado al sistema, por lo que se muestra la pantalla de bienvenida y se ofrece registrarse o loguearse (Figura 46 y Figura 47).

```

28
29     if(ins.estaInstalado()==false){
30         setContentView(R.layout.activity_guibienvenida);
31         this.botonConectarse=(Button) findViewById(R.id.botonConectarse);
32         this.botonRegistrarse=(Button) findViewById(R.id.botonRegistrarse);
33         botonRegistrarse.setOnClickListener(new OnClickListener() {
34             @Override
35             public void onClick(View v) {
36                 registrarse();
37             }
38         });
39         botonConectarse.setOnClickListener(new OnClickListener() {
40             @Override
41             public void onClick(View v) {
42                 loggin();
43             }
44         });
45     }else{
46         System.out.println("instaladooooo");
47         user=ins.getUser();
48         pass=ins.getPass();
49         Intent i=new Intent (this,GUIConversacion.class);
50         i.putExtra("user", user);
51         i.putExtra("pass", pass);
52         startActivity(i);
53     }
54 }
```

Figura 46: comprobar instalación. GUIBienvenida.java

```

31     private boolean comprobarInstalacion(){
32         AgenteFichero af=new AgenteFichero(this.context);
33         String ficheroInstalacion=af.leerFicheroMemoriaInterna(Rutas.getRuta("instalacion"));
34         try{
35             if(ficheroInstalacion.equals(null)){
36                 this.instalado=false;
37             }else{
38                 this.instalado=true;
39             }
40         }catch(Exception e){
41             e.printStackTrace();
42             System.err.print("El fichero de instalacion no existe");
43             //cuando el fichero no existe para que se quede como false.
44             //seguro q hay una manera mejor de hacerlo...
45             //creo que no salta la excepcion aunque no exista asiq esto casi q sobra
46         }
47         return instalado;
48     }
```

Figura 47: comprobarInstalacion. Instalacion.java

En caso de que exista el fichero carga el usuario y la contraseña almacenados, y realiza la acción de login con esos datos. En caso contrario, se nos mostrará la interfaz GUILogin o GUIRegistro respectivamente, para solicitarnos tal información. Una vez hecho el login o el registro, se almacena en el fichero de instalación “instalacion.cnf” y se lanza la interfaz GUIConversacion.

Para realizar la acción de login o registro, al ser una operación a nivel de red, debemos hacerla en un hilo en segundo plano, para no mantener bloqueado el hilo principal mientras se ejecuta. Para ello, Android nos provee de una clase muy útil llamada AsyncTask [34]. Esta clase nos permite crear clases hijas que heredan de ella las cuales se ejecutan en segundo plano, sin interferir con el hilo principal de ejecución. Es una herramienta muy cómoda y útil para la gestión de hilos secundarios de ejecución.

Al hilo de ejecución de la actividad de login o registro, se le debe poner un tiempo límite tras el cual se dará el intento de conexión por fallido. En este caso le hemos marcado 5000 milisegundos máximos (Figura 48).

```
137         showProgress(true);
138         mAuthTask = new UserLoginTask();
139         mAuthTask.execute((Void) null);
140         try {
141             try {
142                 if(mAuthTask.get(5000, TimeUnit.MILLISECONDS)){
143                     Instalacion install=new Instalacion(this);
144                     install.instalar(this.mTfno, this.mPassword);
145                     Intent i=new Intent(this,GUIConversacion.class);
146                     i.putExtra("user", this.mTfno);
147                     i.putExtra("pass", this.mPassword);
148                     startActivity(i);
149                 }else{
150                     mPasswordView.setError(getString(R.string.error_incorrect_password));
151                     mPasswordView.requestFocus();
152                 }
153             }
154         } catch (TimeoutException e) {
155             this.errorLogueo.setVisibility(View.VISIBLE);
156             mAuthTask.cancel(true);
157             e.printStackTrace();
158         }
159         mAuthTask=null;
160         showProgress(false);
```

Figura 48: login. GUILogin.java

La Figura 49 muestra el funcionamiento de una clase que hereda de asyntask, en este caso es la clase UserLoginTask, que es la encargada de loguear al usuario en el sistema.

```
209     public class UserLoginTask extends AsyncTask<Void, Void, Boolean> {
210         private GestorCredencial credenciales = null;
211         @Override
212         protected Boolean doInBackground(Void... params) {
213             this.credenciales=new GestorCredencial(mTfno,mPassword);
214             boolean res=credenciales.enviar();
215             return res;
216         }
217     }
218
219     @Override
220     protected void onPreExecute() {
221         super.onPreExecute();
222         showProgress(true);
223     }
224
225     @Override
226     protected void onPostExecute(final Boolean success) {
227         mAuthTask = null;
228         showProgress(false);
229
230         if (success) {
231             finish();
232         } else {
233             showProgress(false);
234             mPasswordView
235                 .setError(getString(R.string.error_incorrect_password));
236             mPasswordView.requestFocus();
237         }
238     }
239 }
240
241     @Override
242     protected void onCancelled() {
243         mAuthTask = null;
244         showProgress(false);
245     }
246 }
247
248 }
```

Figura 49: AsyncTask. UserLoginTask.java

El sistema de envío de credenciales (usuario+password) para loguearse o para registrarse funciona de forma muy parecida al sistema de envío de mensajes implementado en el ciclo 1. La interfaz crea unas credenciales, las cuales envía utilizando una clase gestora a través de un hilo de ejecución en segundo plano AsyncTask (Figura 50).

```
38     public boolean enviar(){
39         boolean res=false;
40         String cad = "";
41         String credecialesCodificadas=codificar();
42         Envio env=new Envio(credecialesCodificadas);
43         URL direccion = null;
44         try {
45             direccion = new URL(Rutas.getRuta("server") + Rutas.getRuta("log"));
46             env.enviar(direccion,"mensaje");
47             cad=env.getRespuesta();
48         }catch(Exception e){
49             e.printStackTrace();
50         }
51
52         try{
53             if(cad.equals("ok")){
54                 res=true;
55             }
56         }catch(NullPointerException e){
57             e.printStackTrace();
58         }
59         return res;
60     }
```

Figura 50: envio de credenciales. GestorCredenciales.java

Planificación:

Requisitos del cliente:

Requisito	Pendiente	En Proceso	Completo
RF C1			X
RF C2	X		
RF C3	X		
RF C4	X		
RF C5	X		
RF C6	X		
RF C7		X	
RF C8	X		

Requisitos del servidor

Requisito	Pendiente	En Proceso	Completo
RF S1		X	
RF S2	X		
RF S3	X		
RF S4	X		

Una vez finalizado este ciclo, el proyecto tiene un tamaño considerable, y es hora de analizar el estado de este y los requisitos que cumple

Por un lado, la instalación, login y registro en el cliente está completa, y la gestión de usuarios (con persistencia) en el servidor también.

Las rutas de los ficheros, y las URL's del servidor ya no están en el código, sino que se leen de un fichero RAW llamado “rutas”

El cliente no tiene certeza de que los mensajes que envía llegan correctamente al servidor.

El diseño arquitectónico de las dos aplicaciones va tomando forma y será la base para las sucesivas iteraciones (Figura 51 y Figura 52)

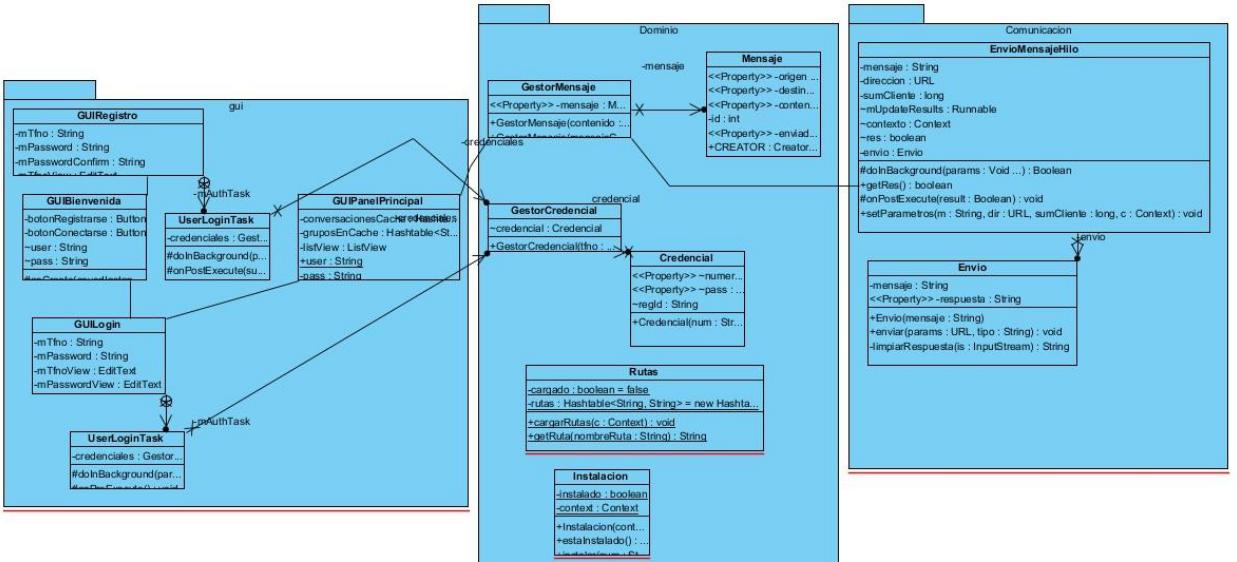


Figura 51: Diagrama de clases UML cliente

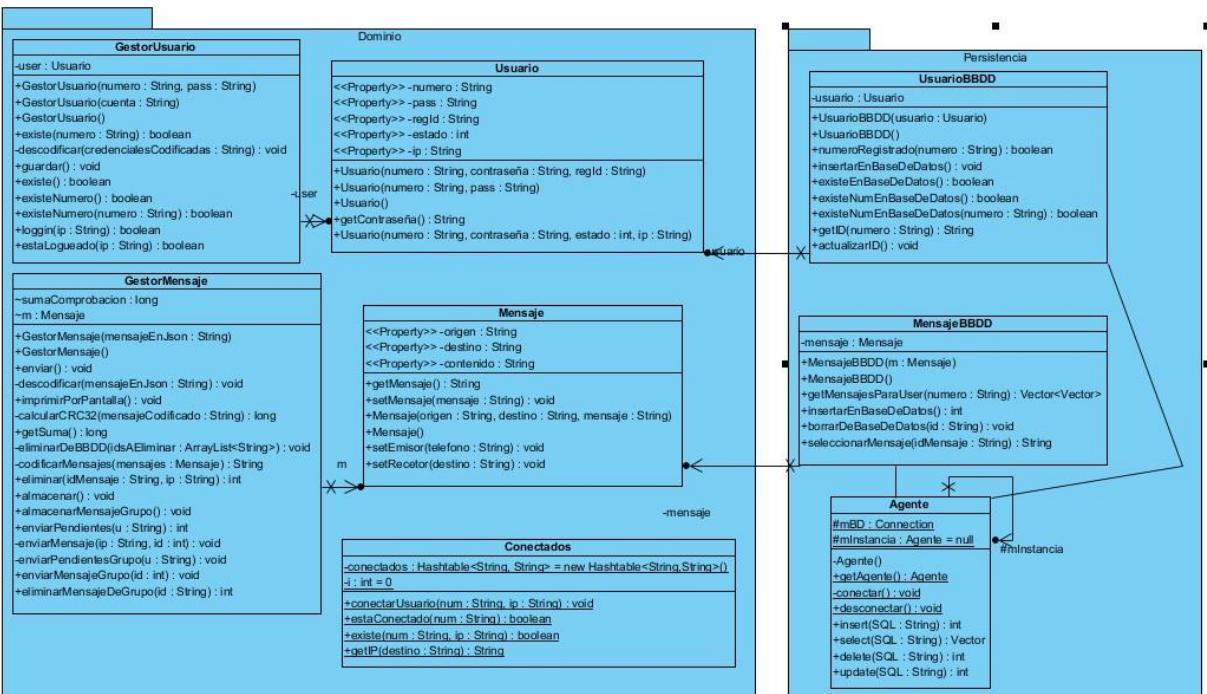


Figura 52: Diagrama de clases UML servidor

4.2.3. Ciclo 3: Mejora en el envío de mensajes.

Determinar Objetivos

Esta tercera iteración, intenta abarcar algo menos de funcionalidad que el anterior. A la vista del usuario, no representa ningún cambio significativo. En este ciclo los requisitos son:

- El cliente debe tener constancia y estar seguro de que el mensaje ha sido recibido correctamente por el servidor.
- El servidor, además de imprimirlas, debe almacenar en su base de datos todos los mensajes que le llegan.
- No hay que añadir ninguna interfaz gráfica más al cliente.
- El sistema de envío de mensajes debe ser el mismo.

Análisis de riesgos

Para almacenar los mensajes en la base de datos, se ha pensado en incluir un atributo ID auto incrementable, así no habrá problemas al insertar dos mensajes de igual contenido y participantes, lo que suponía un riesgo bastante alto.

Para la confirmación de que el mensaje se ha enviado correctamente, se ha pensado utilizar la función HASH para comprobar la correcta recepción de los mensajes, ya que es una función muy extendida y estandarizada (Figura 53).

```
155     private long calcularCRC32(String mensajeCodificado) {  
156         byte bytes[] = mensajeCodificado.getBytes();  
157         Checksum checksum = new CRC32();  
158         checksum.update(bytes, 0, bytes.length);  
159         long checksumValue = 0;  
160         try{  
161             checksumValue = checksum.getValue();  
162         }catch(NumberFormatException e){  
163             e.printStackTrace();  
164         }  
165         return checksumValue;  
166     }
```

Figura 53: calcularCRC32. GestorMensajes.java

El proceso de comprobación consiste en calcular el HASH del mensaje antes de enviarlo, al enviarlo, el servidor nos responde con el HASH del mensaje calculado por él mismo. Si coinciden ambos HASH el mensaje se ha enviado correctamente.

Desarrollar, Verificar y Validar:

En este ciclo tenemos dos objetivos: implementar un sistema que nos asegure que los mensajes han sido recibido correctamente por el servidor, y ampliar la persistencia en el servidor para que almacene los mensajes que recibe.

Para el primer objetivo se ha decidido utilizar la función Hash, y comprobar que el Hash que el servidor calcula del mensaje que recibe es el mismo que el que calcula el cliente antes de enviarlo. La siguiente máquina de estados representa lo que se desea implementar:

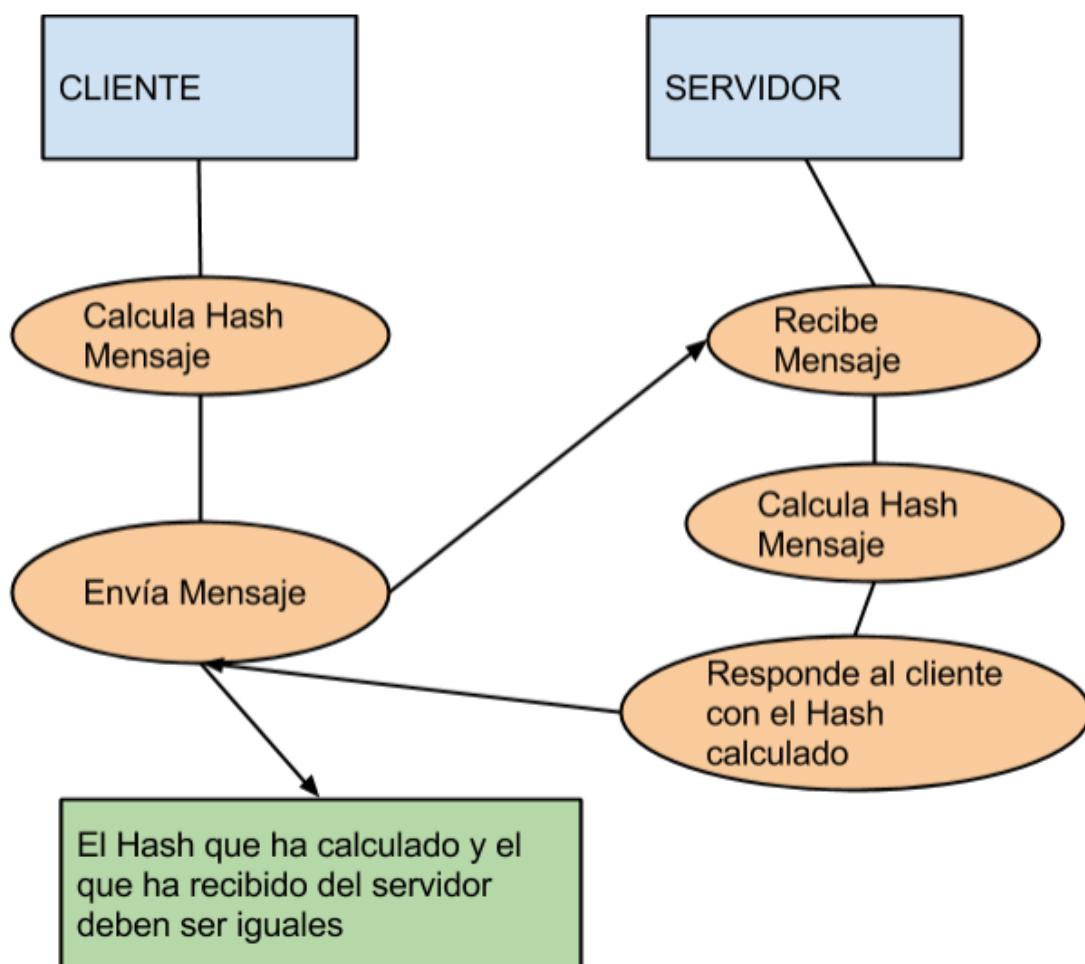


Figura 54: Máquina de estados para la comprobación de envío. Concepto

La idea es que el cliente calcule el Hash del mensaje antes de enviarlo, y que el servidor al recibirla vuelva a calcularlo, y responda al cliente indicándole este nuevo Hash. El cliente comprobará si el Hash recibido por el servidor y el hash que él mismo había calculado antes de enviar el mensaje es igual. En tal caso se aceptará el mensaje como enviado correctamente.

Este proceso se ilustra en las Figuras 55, 56 y 57

```
76     public boolean enviar(Context contexto,int intento,int tipo){  
77         boolean enviado=false;  
78         String mensajeCodificado=codificar();  
79         Long sumaComprobacion=calcularCRC32(mensajeCodificado);  
80         EnvioMensajeHilo env=new EnvioMensajeHilo();  
81         URL direccion = null;  
82         try {  
83             env.enviar(direccion, sumaComprobacion, tipo);  
84             enviado=true;  
85         } catch (Exception e) {  
86             e.printStackTrace();  
87         }  
88     }  
89 }
```

Figura 55: enviar. GestorMensaje.java

```
--  
23         String respuestaServer=envio.getRespuesta();  
24         String cliente=Long.toString(sumCliente);  
25         if(respuestaServer==null){  
26             this.res=false;  
27         }else{  
28             cliente = respuestaServer.replaceAll(" ", " ");  
29             respuestaServer.length();  
30             if(cliente.equalsIgnoreCase(respuestaServer)){  
31                 this.res=true;  
32             }  
33             }  
34         }else{  
35             this.res=false;  
36         }  
37     }  
38 }
```

Figura 56: gestionar respuesta. EnviarMensajeHilo.java

```
13     String texto=null;  
14     try{  
15         texto=request.getParameter("mensaje");  
16         System.out.println(request.getRemoteHost());  
17         GestorMensaje gm=new GestorMensaje(texto);  
18         gm.imprimirPorPantalla();  
19         Long suma = gm.getSuma();  
20         %>  
21         <%=suma%>  
22     }  
23 }
```

Figura 57: recibir mensajes en el Servidor y responder con el Hash. recibir.jsp

Véase en la Figura 57 que la forma que tiene el servidor de responder al mensaje del cliente es integrando la información en el propio HTML resultante de la página (véase sección 2.1.2.). Este mensaje HTML tendrá que ser interpretado por el propio cliente

que envía el mensaje. Para ello se ha diseñado una función que elimina las cabeceras del fichero HTML resultante al hacer el envío del mensaje (Figura 58).

```
66     private String limpiarRespuesta(InputStream is) {  
67         BufferedReader reader = new BufferedReader(new InputStreamReader(is));  
68         StringBuilder sb = new StringBuilder();  
69         String line = null;  
70         try {  
71             while(!(line=reader.readLine()).equals("<body>"));  
72                 while(!(line=reader.readLine()).equals("</body>")){  
73                     sb.append(line);  
74                 }  
75             } catch (IOException e) {  
76                 e.printStackTrace();  
77             } finally {  
78                 try {  
79                     is.close();  
80                 } catch (IOException e) {  
81                     e.printStackTrace();  
82                 }  
83             }  
84         }  
85         return sb.toString();  
86     }  
87 }
```

Figura 58: limpiarRespuesta. Envio.java

Con esto el cliente ya es capaz de saber si el servidor ha recibido el mensaje que le ha enviado, y actuar en consecuencia. En principio si no lo ha recibido no se muestra en la interfaz, pero más adelante se pretende crear una cola de mensajes pendientes de envío, los cuales aún no han sido enviados correctamente.

El segundo objetivo de este ciclo, es aumentar la persistencia de datos del servidor, permitiendo almacenar los mensajes que recibe, así como su origen y destino (el servidor aún no indica un destino concreto, pero lo preparamos para siguientes iteraciones).

Ampliando la base de datos, queda así:



Figura 59: BBDD Servidor. Ciclo 3

usuarios.numero → primary key
mensajes.id → primary key
mensajes.origen → usuario
mensajes.destino → usuario

Los cambios más significativos en el diagrama de clases debido a este aumento de la persistencia serían, de acuerdo a una de las reuniones correspondientes a esta iteración, los que se muestran en la siguiente figura (Figura 60), planificado durante una reunión de control con el director del TFG.

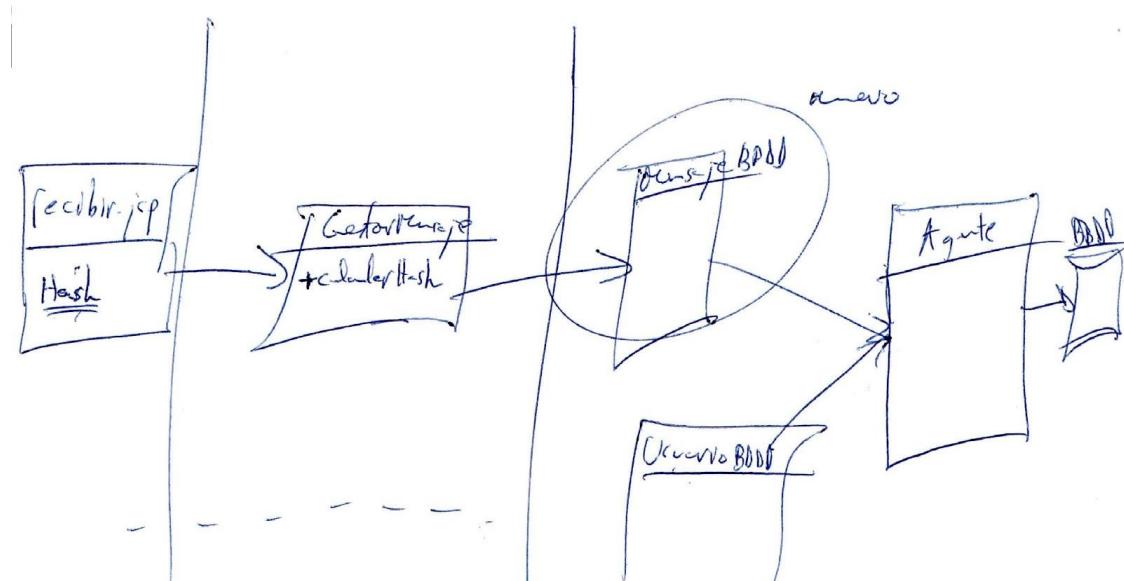


Figura 60: Diagrama de clases Servidor. Ciclo 3

Como se puede apreciar, el sistema de almacenamiento de usuarios y mensajes es muy similar.

Planificación:

Requisitos del cliente:

Requisito	Pendiente	En Proceso	Completo
RF C1			X
RF C2	X		
RF C3	X		
RF C4	X		
RF C5	X		
RF C6	X		
RF C7		X	
RF C8	X		

Requisitos del servidor:

Requisito	Pendiente	En Proceso	Completo
RF S1			X
RF S2	X		
RF S3	X		
RF S4			X

En este último ciclo, no se han implementado nuevos requisitos, sino que se han corregido y mejorado funciones anteriores. A la vista del usuario final este ciclo aporta poco, pero ha sido un ciclo bastante importante en el desarrollo que nos ahorrará muchos futuros problemas.

Recapitulando un poco el sistema:

- El cliente envía mensajes al servidor y comprueba su correcta recepción.
- El servidor almacena los mensajes en su bbdd.
- Instalación, registro, login completo.

Llegados a este punto el servidor está algo más avanzado respecto al cliente, ya que tiene implementada la persistencia y recibe mensajes; y es el cliente el que está limitando la funcionalidad del sistema al no indicar el contacto al que quiere enviar el mensaje, y por ende, el servidor no puede enviárselo al receptor (además de que no está implementada esta función aún).

De modo que en la siguiente iteración se pretende avanzar un poco más en el cliente, y no tocar el servidor. La idea es crear una serie de interfaces y clases para elegir el contacto al que enviar el mensaje. Además se separará cada conversación en función del contacto, de modo que cada contacto tendrá su propia conversación asociada, y no se mezclarán.

4.2.4. Ciclo 4: Selección de contactos. Interfaz principal.

Determinar Objetivos

Este ciclo, a priori parecía sencillo, ya que en principio consistía solo en añadir al mensaje enviado, el número de teléfono del destino. Eso, y seleccionar el número de teléfono es una tarea relativamente sencilla. El problema es que para ello, y de cara a que el usuario encuentre atractivo el entorno, hay que hacer una serie de operaciones que describiremos a continuación:

- Una interfaz con la lista de contactos
- Cargar los contactos del sistema.
- Una interfaz con una lista de conversaciones. Cada conversación es con un contacto distinto.
- Comunicación entre las interfaces.

Lo que parecía un ciclo sencillo, se ha convertido en un cambio bastante significativo en la aplicación. No obstante, al menos en este ciclo, el servidor no sufrirá cambio alguno, sino que todos los cambios se realizan sobre la aplicación cliente.

Análisis de riesgos

La consulta de los contactos del sistema es una operación que en determinados casos puede ser bastante costosa computacionalmente.

Lo ideal sería consultar los contactos existentes en el terminal cada vez que abrimos la interfaz de selección de contacto (para iniciar una nueva conversación), no obstante, en principio, vamos a crear una clase de consulta llamada “Contactos.java”, la cual cargará los contactos del sistema a una tabla de la aplicación con el fin de acceder a ellos más rápido. A la hora de abrir la interfaz de selección de contacto, esta consultará estos datos precargados a la hora de iniciar la ejecución de la aplicación, de manera que cargará la lista de contactos mucho más rápido.

El problema de utilizar este sistema es que para actualizar esta tabla de contactos precargada, hay que salir y volver a iniciar la aplicación. No obstante esto tiene una solución relativamente sencilla que se aplicará en posteriores iteraciones: Añadir un botón en la interfaz de selección de contacto para que cuando el usuario crea oportuno

actualice esta tabla. De esta forma la tabla podrá actualizarse cuando sea necesario, que será pocas veces, y no siempre, ralentizando el flujo de ejecución del programa.

Desarrollar, Verificar y Validar:

Antes de nada es importante comentar que la interfaz GUIConversación.java ya no es llamada al inicio de la ejecución, sino que en su lugar se llama a GUIPanelPrincipal.java.

Al iniciar GUIPanelPrincipal, se cargan los contactos a la clase de consulta Contactos (Figura 61):

```
Cursor cursorNombre = contentResolver.query(CONTENT_URI, null, null, null, null);
if (cursorNombre.getCount() > 0) {
    while (cursorNombre.moveToNext()) {
        String contact_id = cursorNombre.getString(cursorNombre.getColumnIndex(_ID));
        String nombre = cursorNombre.getString(cursorNombre.getColumnIndex(DISPLAY_NAME));
        int hasPhoneNumber = Integer.parseInt(cursorNombre.getString(cursorNombre.getColumnIndex(HAS_PHONE_NUMBER)));
        if (hasPhoneNumber > 0) {
            Cursor cursorNumero = contentResolver.query(PhoneCONTENT_URI, null,
                Phone_CONTACT_ID + " = ?",
                new String[] { contact_id }, null);
            while (cursorNumero.moveToNext()) {
                String numeroTfno = cursorNumero.getString(cursorNumero.getColumnIndex(NUMBER));
                numeroTfno=limpiar(numeroTfno);
                contactos.add(new Contacto(numeroTfno,nombre));
            }
            cursorNumero.close();
        }
    }
    cursorNombre.close();
}
```

Figura 61: Carga de Contactos. Contactos.java

Como algunos contactos contienen espacio, prefijos u otros modificadores, al número obtenido de la agenda le aplicamos un filtro que elimina los prefijos y espacios con el fin de que todos los contactos contengan un número de 9 cifras lo más estándar posible.

Para comenzar una nueva conversación se utiliza el GUIContactos, que extiende de ListActivity y tan solo contiene una lista con los contactos de nuestro sistema, con el fin de seleccionar uno y comenzar una conversación con él (Figura 62).

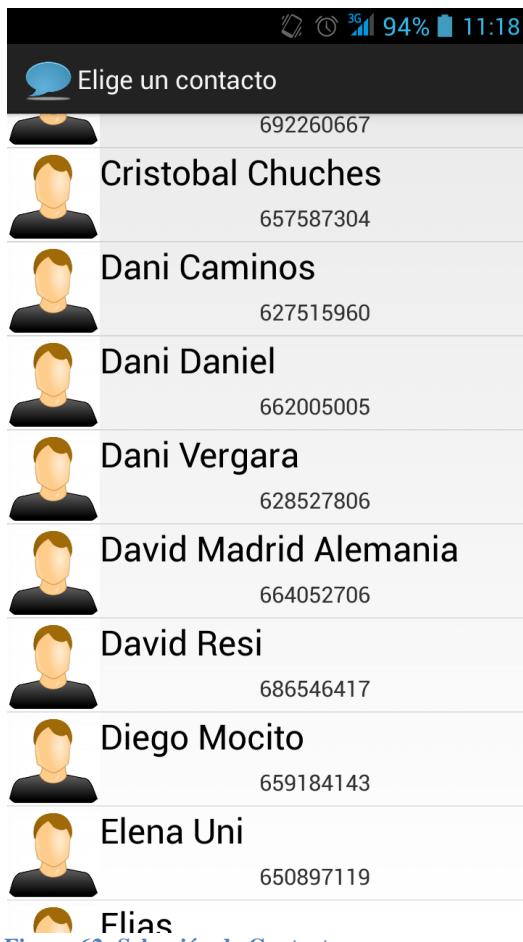


Figura 62: Selección de Contacto

Para mostrar una lista en Android se utilizan los Adapter (Adaptadores), que son clases que heredan de baseAdapter, y nos permiten personalizar la vista de la lista. Para el caso de la lista de contactos, se ha creado el adaptador “ItemAdapter.java”

```
45     @Override
46     public View getView(int position, View convertView, ViewGroup parent) {
47         View rowView = convertView;
48         if (convertView == null) {
49             LayoutInflater inflater = (LayoutInflater) context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
50             rowView = inflater.inflate(R.layout.elemento_lista, parent, false);
51         }
52         TextView vistaNombre = (TextView) rowView.findViewById(R.id.nombre);
53         TextView vistaNumero = (TextView) rowView.findViewById(R.id.numero);
54         Contacto item = this.items.get(position);
55         vistaNombre.setText(item.getNombre());
56         vistaNumero.setText(item.getNumero());
57         return rowView;
58     }
59 }
60 }
```

Figura 63: Adaptador de Contactos. ItemAdapter.java

Como se puede apreciar, el adapter recorre la lista de elementos de la lista, y va asignando su valor a cada uno de los atributos de cada elemento de la lista.

Aunque a priori pueda parecer algo un poco complicado, al final es muy útil tener un adaptador para cada tipo de lista, ya que en caso de necesidad, se pueden modificar las vistas de las listas de forma rápida y segura.

Al seleccionar un contacto, se envía al GUIPanelPrincipal, que crea una conversación con él, y añade esa conversación a la lista.

Para recoger información de una actividad finalizada (en este caso GUIContactos) se ha hecho lo siguiente:

Al abrir la activity GUIContactos de GUIPanelPrincipal, no se abre de forma normal, sino que se le indica que esperamos una respuesta con la función “startActivityForResult” (Figura 64).

```
private void lanzarGUISeleccionContacto() {  
    Intent i=new Intent(this,GUIContactos.class);  
    Contacto c=null;  
    i.putExtra("seleccion", c);  
    startActivityForResult(i,0);  
}
```

Figura 64: lanzar Interfaz de selección de contacto. GUIPanelPrincipal.java

Para recoger el resultado de la selección de GUIContactos, utilizamos la función “onActivityResult”, y aquí hacemos las operaciones necesarias para crear la nueva conversación y añadirla a nuestra lista de conversaciones (Figura 65).

```
@Override  
protected void onActivityResult(int requestCode, int resultCode, Intent data) {  
  
    if ( requestCode == 0 ){  
        if ( resultCode == Activity.RESULT_OK ){  
            Contacto resultado=(Contacto) data.getExtras().get("seleccion");  
  
            for(int i=0;i<recientes.size();i++){  
                if(recientes.get(i).getClave().equals(resultado.getNumero())){  
                    existe=true;  
                    break;  
                }  
            }  
            if(!existe){  
                this.recientes.add(0,new ItemPrincipal("conversacion", resultado.getNombre(),  
                    resultado.getNumero());  
                this.conversacionesCache.put(resultado.getNumero(), new Conversacion());  
                listView.setAdapter(new ListaConversacionesAdapter(this, this.recientes));  
            }  
        }  
    }  
}
```

Figura 65: Recoger resultado de activity. GUIPanelPrincipal.java

Como vemos, las actividades se comunican con un identificador entero, en el caso de la comunicación GUIPanelPrincipal-GUIContactos, se utiliza el identificador 0.

De este modo con la misma función “onActivityResult” podemos recuperar el resultado de otras actividades que estén asociadas, tan solo indicándole otro identificador.

Llegados a este punto, hay un aspecto a tener muy en cuenta. El paso de parámetros de un activity a otro se realiza de forma un tanto diferente que en otros sistemas.

En principio, en envío de un parámetro se realiza con la función `putExtra()` y la recepción del parámetro con `getExtra()`. El problema es que estas funciones no trabajan con objetos normales, sino que hay que *parcelarlos*.

Hacer un objeto parcelable es una especie de serialización muy rápida y cómoda que solo existe en Android. Para parcelar un objeto, este tiene que ser una instancia una clase parcelable, por lo cual debe implementar la interfaz `Parcelable` de Android. Además hay que implementar las funciones de la interfaz `Parcelable`, lo que complica un poco el proceso de envío y recepción de parámetros.

Veamos un ejemplo de *parcelalización* con la clase `Contacto` que es la que necesitamos pasar como parámetro en este caso:

```

6  public class Contacto implements Parcelable{
7      protected String nombre;
8      protected String numero;
9
10     public Contacto(String numero, String nombre) {
11         super();
12         this.nombre = nombre;
13         this.numero = numero;
14     }
15
16     public Contacto(){
17
18     }
19
20     private Contacto(Parcel in){
21         this();
22         readFromParcel(in);
23
24     }
25
26     private void readFromParcel(Parcel in) {
27         this.nombre = in.readString();
28         this.numero = in.readString();
29     }
30
31     @Override
32     public int describeContents() {
33         return 0;
34     }
35
36     @Override
37     public void writeToParcel(Parcel dest, int flags) {
38         dest.writeString(this.nombre);
39         dest.writeString(this.numero);
40     }
41
42     public static final Contacto.Creacion<Contacto> CREATOR = new Contacto.Creacion<Contacto>() {
43         @Override
44         public Contacto createFromParcel(Parcel in) {
45             return new Contacto(in);
46         }
47
48         @Override
49         public Contacto[] newArray(int size) {
50             return new Contacto[size];
51         }
52     };

```

Figura 66: parcelización de una clase. Contacto.java

Como se aprecia en la figura 66, para implementar la interfaz serializable en nuestra clase hay que hacer lo siguiente:

- Crear un constructor vacío.
- Crear un constructor que acepte un Parcel como argumento y que lea este parcel almacenando y guarde los datos en sus atributos de clase. Es una especie de reconstrucción del objeto enviado.

- Una función writeToParcel, que guarde en un objeto Parcel el contenido de los atributos del objeto a enviar.
- Una función estática CREATOR para almacenar arrays del objeto parcelable.

En este caso concreto, para almacenar/recuperar atributos de la clase String se usa readString / writeString, pero se podrían almacenar muchos otros tipos con otras funciones como por ejemplo:

- writeTypedListList(<T> val)
- writeLong(Long val)
- writeFloat(float val)
- writeDouble(double val)

Para más información y consultas de otros tipos se debe consultar el manual de *Android Parcel [35]*

Con estas implementaciones la interfaz principal tendría el siguiente aspecto:



Figura 67: Interfaz principal. Ciclo 4

La interfaz principal también utiliza una lista para mostrar las conversaciones existentes, pero en este caso usa otro adapter “ListaConversacionesAdapter” el cual posiciona los elementos de una forma distinta en la interfaz.

Otro aspecto destacable de esta interfaz es lo que almacena y cómo lo hace.

Por un lado almacena el nombre de usuario (el teléfono) y lo va pasando por parámetros a las actividades que lo van necesitando.

Por otro lado almacena una tabla Hash (conversacionesEnCache), con las conversaciones que tenemos iniciadas. La clave de la tabla será el número de teléfono del contacto.

También almacena un lista con los contactos para los que hay una conversación iniciada (recientes). Esta lista es la que se le pasa al adaptador del ListView para mostrar las conversaciones iniciadas. Tanto la lista recientes, como la tabla “conversacionesEnCache” tendrán la misma longitud siempre.

Una vez controlado todo en GUIPanelPrincipal, pasamos a explicar los cambios realizados en GUIConversacion.

En los ciclos anteriores GUIConversación era el punto de partida de la aplicación, sin embargo ahora solo se lanza cuando abrimos una conversación (Figura 68).

```
@Override  
public void onItemClick(AdapterView<?> parent, View view,  
    int position, Long id) {  
  
    iConversacion=new Intent(getApplicationContext(),GUIConversacion.class);  
    Contacto c=Contactos.findContactoByNumero(((Contacto)listView.getItemAtPosition(position)));  
    iConversacion.putExtra("miNumero", user);  
    iConversacion.putExtra("numeroTfno", c.getNumero());  
    iConversacion.putExtra("nombre", c.getNombre());  
    elegida=conversacionesCache.get(c.getNumero());  
    iConversacion.putExtra("conversacion",elegida);  
    startActivityForResult(iConversacion,1);  
    listView.setAdapter(new ListaConversacionesAdapter(getApplicationContext(), recientes));
```

Figura 68: abrir conversación. GUIPanelPrincipal.java

Ahora a GUIConversación hay que pasarle los datos del destino además de nuestro número, además de la conversación almacenada si es que ya existía. Vemos como a esta actividad le pasamos el identificador 1 para distinguirla de la actividad de selección de contactos.

Los cambios en el la actividad GUIConversacion son mínimos, tan solo hemos se ha cambiado la forma de mostrar los mensajes, que en ciclos anteriores era directamente sobre un textView, y ahora lo hacemos con otra lista, controlada con el adaptador “ConverAdapter”.

Además, al iniciar la actividad, debe cargar la conversación que le estamos pasando desde GUIPanelPrincipal en esta lista (Figura 69).

```
private void actualizarGUI() {
    Context aux=getBaseContext();
    listView.setAdapter(new ConverAdapter(aux, this.conversacion.getConversacion(),this.miNumero));
    listView.setSelection(listView.getCount() - 1);
}

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_conversacion);
    conversacion=new Conversacion();
    botonEnviar = (Button) findViewById(R.id.button1);
    //cuadroConversacion=(TextView) findViewById(R.id.textView1);
    cuadroMensaje=(EditText)findViewById(R.id.editText2);
    Bundle bundle=this.getIntent().getExtras();
    this.nombre=bundle.getString("nombre");
    this.numero=bundle.getString("numeroTfno");
    this.miNumero=bundle.getString("miNumero");
    this.conversacion=bundle.getParcelable("conversacion");
    this.listView = (ListView) findViewById(android.R.id.list);
    this.setTitle(this.nombre);
    actualizarGUI();
}
```

Figura 69: creación y actualización de conversación. GUIConversación.java

Con este último cambio, hemos terminado de cumplir los requisitos planificados para este ciclo.

Pese al insignificante cambio a nivel funcional que implicaba (tan solo queríamos poder modificar el destino del mensaje), los cambios a nivel estructural han sido bastante significativos, y la estructura del proyecto ha crecido bastante.

Planificación

Requisitos del cliente:

Requisito	Pendiente	En Proceso	Completo
RF C1			X
RF C2			X
RF C3			X
RF C4	X		
RF C5	X		
RF C6	X		
RF C7			X
RF C8	X		

Requisitos del servidor:

Requisito	Pendiente	En Proceso	Completo
RF S1			X
RF S2	X		
RF S3	X		
RF S4			X

Tras este ciclo, se podría decir que se ha completado la mitad de la comunicación entre dos clientes. Los clientes pueden enviar mensajes a cualquier contacto, pero no pueden recibir los que otros contactos le envían, así que la comunicación sigue estando incompleta.

En el siguiente ciclo, se va a intentar completar esta comunicación, ya que el objetivo principal será que los clientes reciban mensajes los mensajes que otros clientes le envían.

4.2.5. Ciclo 5: Recepción de mensajes

Determinar Objetivos

En este ciclo, el objetivo principal es terminar el procedimiento básico de envío/recepción de mensajes. En ciclos anteriores se ha conseguido enviar mensajes al servidor indicándole el destino, pero hasta ahora, el servidor no tiene forma de enviarle esos mensajes al destino.

El objetivo principal es implementar un sistema de recepción de mensajes en el cliente que permita al servidor enviarle los mensajes que tiene almacenados para él, verificando su origen y añadiéndolo en función de este a su conversación correspondiente

- Recibir mensajes del servidor
- Analizar origen del mensaje
- Añadir a conversación existente, o crear conversación si no existe aún
- Actualizar las interfaces gráficas con los nuevos mensajes recibidos

Análisis de riesgos

Para llevar este proceso de recepción a cabo se han pensado varias alternativas:

- Por un lado se pensó en implementar una escucha en un puerto determinado, de forma que el servidor envíe los mensajes al cliente a través de un socket conectado a ese puerto

-Por otro lado, se pensó en un sistema que realice peticiones HTTP al servidor con una frecuencia de tiempo determinada, y el servidor envíe los mensajes pendientes al cliente en la respuesta HTML de esa petición HTTP.

En principio, y aprovechando el uso de JSP y JSON, se ha decidido usar la segunda opción. No obstante es una opción que no termina de convencer.

Para llevar a cabo la recepción de mensajes mediante peticiones HTTP hay que implementar una serie de elementos:

1. Un servicio (Service), que actúe en segundo plano y cada vez que reciba un aviso, envíe una petición HTTP al servidor para recibir sus mensajes pendientes.
2. Una alarma programada, que cada x unidades de tiempo, avise al servicio para que realice la petición HTTP

3. Un receptor de alarma, que reciba la señal de la alarma y avise al service para que realice la petición HTTP.

4 Sistemas de control para el servicio, para casos en los que no haya conexión de red, no se reciban mensajes, etc.

Como vemos, la opción elegida implica una serie de elementos que posiblemente nos ahorraríamos utilizando sockets. Así que, pese a implementar esta opción, no se descarta modificar el sistema y utilizar la primera opción en futuras iteraciones.

Desarrollar, Verificar y Validar:

En primer lugar, implementamos un servicio que actué en segundo plano. Este hilo secundario será el encargado de enviar la petición al servidor, a través de un gestor de mensajes (Figura 70).

```
4  public class ReceptorMensajes extends IntentService {
5      private String user;
6      public ReceptorMensajes() {
7          super("receptor-mensajes");
8      }
9
10     @Override
11     protected void onHandleIntent(Intent intent) {
12         this.user=(String) intent.getExtras().get("user");
13         GestorMensaje gm=new GestorMensaje(GUIPanelPrincipal.actualizador);
14         gm.getMensajes(user,this.getBaseContext());
15
16     }
17
18 }
```

Figura 70: ReceptorMensajes.java

Esta clase extiende directamente de IntentService, un servicio que se ejecuta en segundo plano en Android. Cada vez que se llama al servicio, se ejecuta la función “onHandleIntent”.

El receptor de mensajes no envía las peticiones directamente, sino que se apoya en un gestor de mensajes para recibir los mensajes, analizarlos, y realizar las operaciones pertinentes con ellos (Figura 71).

```

public void getMensajes(String numero, Context contexto){
    Envio e=new Envio(numero);
    String mensajesCodificados="";
    try {
        e.enviar(new URL(Rutas.getRuta("server") + Rutas.getRuta("get")), "mensaje");
        mensajesCodificados=e.getRespuesta();
    } catch (MalformedURLException e1) {
        e1.printStackTrace();
    }

    //comprobamos que hay mensajes
    if(mensajesCodificados!=null){
        if(!mensajesCodificados.equals("no logueado")){
            List<Mensaje> listaMensajes=decodificarVarios(mensajesCodificados);
            try{
                if(listaMensajes.size()>0){

                    for(int i=0;i<listaMensajes.size();i++){
                        Mensaje m=listaMensajes.get(i);
                        GUIPanelPrincipal.actualizador.actualizar(m,1);
                    }
                }
            }catch(NullPointerException ex){
                System.out.println("Fallo de conexion");
            }
        }
    }else{
        Toast.makeText(contexto, "se ha perdido la conexion con el servidor", Toast.LENGTH_LONG).show();
    }
}

```

Figura 71: Envío de petición de recepción de mensajes. GestorMensaje.java

En este punto, el sistema se complica un poco, así que se intentará detallar lo máximo posible apoyándonos en el mayor número de capturas de código posible.

El proceso es el siguiente:

1. Se envía una petición al servidor para que se nos envíen los mensajes que tenemos. Para ello le indicamos nuestro número de teléfono. Esta acción se realiza en la clase Envío.
2. Una vez recibidos los mensajes, la clase Envío limpia las respuestas, eliminándole las cabeceras y datos que no sean el propio mensaje. Esta acción también se realiza en la clase Envío.
3. Se descodifican los mensajes, es decir, se transforman del formato JSON que utiliza el servidor para enviarnos el mensaje, en un objeto que instancia la clase Mensaje.
4. Para cada mensaje recibido, se envía una notificación al objeto Actualizador de “GUIPanelPrincipal” y aquí es donde se complica un poco el mecanismo.

Todo este proceso se está ejecutando en segundo plano, es decir en un hilo de ejecución que no es el principal, donde se ejecuta la interfaz gráfica. La idea es actualizar la interfaz gráfica del programa con el nuevo mensaje recibido de forma automática, así que la clase Actualizador realizará este proceso.

El gran problema de todo esto es tan sencillo de explicar cómo difícil de solucionar. El sistema operativo Android tiene una restricción en el sistema: La interfaz gráfica solo puede modificarse desde el hilo de ejecución principal, en caso contrario, nos lanzará una excepción.

La propia API de Android provee de un sistema para realizar este tipo de operaciones “runonUiThread”, no obstante no ha funcionado tan bien como se esperaba.

Analizando el problema se concluye que la solución pasa por que el hilo secundario notifique al hilo principal y este sea el que actualice la interfaz gráfica.

Para comunicar dos hilos de ejecución se ha optado por una solución clásica y manual: Handle (Manejador).

Un Handle es un mecanismo de comunicación entre dos hilos de ejecución distintos. Se utilizará un Handle para comunicar el hilo secundario con el principal y poder actualizar la interfaz gráfica. Para llevar este proceso a cabo se necesitan varios elementos.

En primer lugar escribimos un Runnable que será accedido por el Handle y ejecutará un código en el hilo principal (Figura 72).

```
final Runnable mUpdateResults = new Runnable() {
    @Override
    public void run() {
        //código a ejecutar en el hilo principal
    }
};
```

Figura 72: Runnable de actualización

En segundo lugar creamos el handler que más tarde utilizaremos para acceder al hilo principal (Fig. 73):

```
final Handler mHandler = new Handler();
```

Figura 73: creación de Handle

Después, creamos un objeto actualizador, e insertamos el manejador que hemos creado y el Runnable al que debe acceder (Fig 74):

```
actualizador=new Actualizador(this.conversacionesCache);
actualizador.addObserver(mHandler, mUpdateResults);
```

Figura 74: Creación del actualizador

Por último, la clase Actualizador, será una clase normal, que contendrá una lista de Runnable y una lista de Manejadores, aunque en principio solo existirá uno de cada tipo. El actualizador tendrá una función actualizar (Figura 75):

```
public void actualizar(Mensaje m,int opcion){
    boolean enCache=false;
    switch(opcion){
        case 1://entrada
            enCache=conversacionesCache.containsKey(m.getOrigen());
            if(enCache){
                conversacionesCache.get(m.getOrigen()).setMensaje(m);
            }else{
                Conversacion nueva=new Conversacion();
                nueva.setMensaje(m);
                conversacionesCache.put(m.getOrigen(), nueva);
            }
            this.numeroRepcion=m.getOrigen();
            break;

        for(int i=0;i<this.mHandlerList.size();i++){
            this.mHandlerList.get(i).post(this.mUpdateResultsList.get(i));
        }
    }
}
```

Figura 75: actualizar. Actualizador.java

La función actualizar, insertará el mensaje recibido en su conversación correspondiente, o la creará en caso de que no exista, y, además, notificará al hilo principal de lo ocurrido, y será el hilo principal el que actualice la interfaz gráfica en consecuencia.

Otro detalle a tener en cuenta es que no sólo hay que actualizar la interfaz principal, sino que es posible que nos encontremos en una interfaz de conversación y sea necesario actualizar esa interfaz en tiempo real.

Para notificar a la interfaz de conversación en caso de que se encuentre abierta, se ha decidido que sea la interfaz principal el que la notifique, ya que esta ya tiene la conversación actualizada. El problema es que para notificar a la interfaz de conversación, tendremos que hacerlo mediante un Intent, y comunicar este nuevo intent con la instancia de GUIConversacion en ejecución. Para ello se utiliza la función “onNewIntent()” heredada de Activity (Figura 76).

```

49     protected void onNewIntent(Intent intent) {
50         super.onNewIntent(intent);
51         if ("action.action.myactionstring".equals(intent.getAction())) {
52             this.conversacion=(Conversacion) intent.getExtras().get("conversacionActualizada");
53             actualizarGUI();
54         }
55     }
56     private void actualizarGUI() {
57         Context aux=getBaseContext();
58         listView.setAdapter(new ConverAdapter(aux, this,conversacion.getConversacion(),this.miNumero));
59         listView.setSelection(listView.getCount() - 1);
60     }
61 }
```

Figura 76: actualización de GUIConversacion. GUIConversacion.java

Así desde GUIPanelPrincipal podremos lanzarle un nuevo Intent a una activity ya creada, y además pasarle los datos actualizados de su conversación (Figura 77).

```

if(enConversacion && numeroEnConversacion.equals(c.getNumero())){
    c.quitarPendientes();

    Intent intent = new Intent("action.action.myactionstring");
    intent.addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP);
    Conversacion cActualizada=conversacionesCache.get(c.getNumero());
    intent.putExtra("conversacionActualizada",cActualizada);
    startActivity(intent);
}else{
}
```

Figura 77: Control de conversación abierta. GUIPanelPrincipal.java

Para controlar que GUIConversación está abierto utilizamos una variable booleana que se activa cuando entramos en una conversación, y se desactiva al salir. Además se guarda en la variable “numeroEnConversacion” el número del contacto con el que tenemos abierta la conversación.

Por último, para que GUIConversacion acepte el nuevo intent, debemos añadirle un filtro en el archivo manifest, para identificar el nuevo Intent (Figura 78).

```

<activity
    android:name="com.laguna.talkme.gui.GUIConversacion"
    android:label="@string/app_name" >
    <intent-filter>
        <action android:name="action.action.myactionstring" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
<activity
```

Figura 78: Filtro para la recepción de nuevos Intents en GUIConversación. Manifest.xml

Planificación:

Requisito	Pendiente	En Proceso	Completo
RF C1			X
RF C2			X
RF C3			X
RF C4	X		
RF C5	X		
RF C6	X		
RF C7			X
RF C8			X

Requisitos del servidor

Requisito	Pendiente	En Proceso	Completo
RF S1			X
RF S2			X
RF S3			X
RF S4			X

Con estas mejoras, el sistema es capaz de recibir mensajes, y actualizar las interfaces gráficas del sistema en tiempo real, desde hilos de ejecución secundarios, y sin interferir en el hilo principal.

Con esto se ha completado la base de la comunicación entre dos usuarios. En siguientes iteraciones se mejorará esta comunicación, así como su presentación, pero partiendo de esta base.

4.2.6. Ciclo 6: Persistencia en el cliente

Determinar Objetivos

En esta iteración el objetivo principal es almacenar de forma persistente los mensajes recibidos, enviados, y conversaciones, de modo que al salir y entrar a la aplicación se puedan recuperar las conversaciones mantenidas con anterioridad. Además se pretenderá incluir unos registros temporales que almacenarán mensajes que no han podido ser enviados (por falta de conexión) a la espera de poder ser enviados. Además, aprovecharemos para almacenar también los contactos del sistema, de modo que no tengamos que acceder a ellos que entremos en la interfaz de selección de contacto.

Análisis de riesgos

Para la persistencia de datos en el cliente se ha optado por una base de datos SQLite. Las razones de la elección son las siguientes:

- Android incluye un soporte nativo para una base de datos SQLite.
- No necesita un servidor para ejecutarse
- Todos la base de datos se encuentra contenida en un solo archivo
- Es muy portable
- Utiliza código libre, de modo que es gratuito su uso

Desarrollar, Verificar y Validar:

Para implementar la base de datos en nuestro cliente, lo primero que debemos hacer es definir las tablas independientemente del sistema de bases de datos a utilizar (Figura 79).

Usuarios	Mensajes	MensajesPendientes
numero: varchar nombre: varchar	id: integer origen: text destino: text contenido: text enviado: text	id: integer (autoIncrement) origen: text destino: text contenido: text

Figura 79: BBDD Cliente (Ciclo 6)

Usuarios.numero → primary key

Mensajes.id → primary key

MensajesPendientes.id → primary key

La tabla usuarios almacenará todos los números de la agenda del terminal, y su nombre. La clave principal es el número, por lo que no se podrá tener dos contactos con el mismo número de teléfono.

La tabla mensajes almacena los mensajes tanto los enviados como los recibidos. Introducimos un campo ID como clave principal ya que si usaramos origen-destino-contenido como clave principal, un usuario A no podría enviar a un usuario B dos mensajes con el mismo contenido.

La tabla MensajesPendientes almacena únicamente los mensajes que no han podido ser enviados por falta de conexión o un error en el sistema, para una vez el sistema se recupere o se conecte a internet, pueda acceder a los mensajes pendientes rápidamente y enviarlos. En ese momento se eliminan de esta tabla y se actualiza la tabla mensajes poniendo el valor de su campo enviado a verdadero. La clave principal es ID por la misma razón que la tabla Mensajes.

Una vez diseñada la base de datos se implementa en nuestro sistema.

En primer lugar diseñamos una clase agente, con un patrón *Singleton* que nos asegure que solo existirá una instancia que acceda a la base de datos. Esta clase extiende de “SQLiteOpenHelper” una clase que nos ofrece Android para facilitar el acceso y modificación de bases de datos (Figura 80).

```

public class AgenteBBDD extends SQLiteOpenHelper {

    private static AgenteBBDD instancia;

    private AgenteBBDD(Context contexto, String nombre,
                       CursorFactory factory, int version) {
        super(contexto, nombre, factory, version);
    }

    public static AgenteBBDD getInstancia(Context contexto){
        if(instancia==null){
            instancia=new AgenteBBDD (contexto, "DBUsuarios", null, 1);
        }
        return instancia;
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        //Se ejecuta la sentencia SQL de creación de la tabla
        db.execSQL("CREATE TABLE \"usuarios\" (\"numero\""
                   + " VARCHAR PRIMARY KEY NOT NULL UNIQUE , \"nombre\" VARCHAR)");
        db.execSQL("CREATE TABLE \"mensajes\" (\"id\" INTEGER NOT NULL , \"origen\""
                   + " VARCHAR NOT NULL , \"destino\" VARCHAR NOT NULL , \"contenido\" VARCHAR NOT NULL ,
                   \"enviado\" INTEGER NOT NULL, PRIMARY KEY (\"id\", \"origen\", \"destino\"))");
        db.execSQL("CREATE TABLE \"mensajesPendientes\" (\"id\""
                   + " INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE , \"origen\" TEXT NOT NULL
                   check(typeof(\"origen\") = 'text') , \"destino\" TEXT NOT NULL check(typeof(\"destino\") = 'text') ,
                   | \"contenido\" TEXT NOT NULL check(typeof(\"contenido\") = 'text') )");
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int versionAnterior, int versionNueva) {
        db.execSQL("DROP TABLE IF EXISTS Usuarios");
    }
}

```

Figura 80: Agente de bases de datos del cliente. AgenteBBDD.java

Esta clase será la única que tenga contacto directo con el sistema de base de datos. Como vemos aquí se crean las tablas para la BBDD. En futuras iteraciones cuando se necesiten insertar más tablas en la base de datos, se incluirán aquí de igual manera que las actuales tablas.

Ahora, para cada clase de conocimiento, la cual implica una tabla en la base de datos, creamos una clase Agente, que transformará el objeto de la clase, en una sentencia SQL.

A continuación se muestra como ejemplo el adaptador para mensajes (MensajeBBDD), pero la estructura sería la misma para la tabla-clase contactos, y para los mensajes temporales.

```

public class MensajeBBDD {
    private SQLiteDatabase db;
    private AgenteBBDD dbHelper;

    public MensajeBBDD(Context context) {
        dbHelper = AgenteBBDD.getInstancia(context);
    }

    public void open() {
        db = dbHelper.getWritableDatabase();
    }

    public void close() {
        dbHelper.close();
    }

    public void insertarMensaje(Mensaje m) {
        ContentValues values = new ContentValues();
        values.put("id", m.getID());
        values.put("origen", m.getOrigen());
        values.put("destino", m.getDestino());
        values.put("contenido", m.getContenido());
        if(m.isEnviado()){
            values.put("enviado", "1");
        }else{
            values.put("enviado", "0");
        }

        try{
            db.insert("mensajes", null, values);
        }catch(Exception e){
            e.printStackTrace();
        }
    }

    public List<Mensaje> getAllMensajes() { ... }

    private Mensaje cursorToMensaje(Cursor cursor) { ... }

    public void actualizarMensaje(Mensaje mensaje) { ... }
}

```

Figura 81: Adaptador de mensajes a base de datos. MensajeBBDD.java

Como se aprecia, existe una función para insertar el mensaje, otra para recuperar los mensajes de la base de datos, y otra para actualizar el contenido de un mensaje. La única función que se podría destacar es “cursorToMensaje()”, cuya función es transformar la información procedente de la base de datos leída mediante un cursor, en un objeto de la clase Mensaje. Obviamente esta función cambiará en los adaptadores de las demás tablas siendo “cursorToTemporal()” y “cursorToContacto()” (Figura 82).

```

private Mensaje cursorToMensaje(Cursor cursor) {
    Mensaje mensaje = new Mensaje();
    mensaje.setID(Integer.parseInt(cursor.getString(0)));
    mensaje.setOrigen(cursor.getString(1));
    mensaje.setDestino(cursor.getString(2));
    mensaje.setContenido(cursor.getString(3));
    if(cursor.getString(4).equals("0")){
        mensaje.setEnviado(false);
    }else{
        mensaje.setEnviado(true);
    }
    return mensaje;
}

```

Figura 82: de cursor a mensaje. MensajeBBDD.java

Por último actualizamos las funciones getMensajes y enviar de GestorMensaje.java insertando el mensaje en la base de datos cuando lo envía al servidor o lo recibe (Figura 83):

```

if(listaMensajes.size()>0){
    MensajeBBDD mbd=new MensajeBBDD(contexto);
    mbd.open();
    for(int i=0;i<listaMensajes.size();i++){
        Mensaje m=listaMensajes.get(i);
        GUIPanelPrincipal.actualizador.actualizar(m,1);
        m.setEnviado(true);
        mbd.insertarMensaje(m);
    }
    mbd.close();
}else{
}

```

Figura 83: getMensajes modificado. GestorMensaje.java

```

if(intento==1){
    GUIPanelPrincipal.actualizador.actualizar(this.mensaje,2);
    MensajeBBDD mbd=new MensajeBBDD(contexto);
    mbd.open();
    mbd.insertarMensaje(this.mensaje);
    mbd.close();
}else{
    GUIPanelPrincipal.actualizador.actualizar(this.mensaje,3);
    MensajeBBDD mbd=new MensajeBBDD(contexto);
    mbd.open();
    mbd.actualizarMensaje(this.mensaje);
    mbd.close();
}
return enviado;

```

Figura 84: enviar modificado. GestorMensaje.java

Con esto se quedan guardados todos los mensajes, tanto salientes como entrantes, en la base de datos. A continuación se describe cómo se recuperan al iniciar la ejecución del programa.

Cuando iniciamos el cliente, antes de recibir mensajes se deben recuperar los mensajes existentes, para insertar los mensajes entrantes en sus correspondientes conversaciones. Para lanzamos la función “cargarConversacionesBBDD()” en la etapa “onCreate()”

La función cargarConversacionesBBDD utiliza a su vez una clase que hemos creado exclusivamente para la carga de las conversaciones de la base de datos (CargadorConversaciones.java).

Esta clase se encarga exclusivamente de acceder mediante los adaptadores de mensajes a la base de datos, recuperar todos los mensajes, y construir las conversaciones almacenadas. Esto lo hace una única función (Figura 85)

```
public void cargarDeBBDD(Context c, Hashtable<String, Conversacion> conversaciones){
    //Hashtable<String, Conversacion> conversaciones=new Hashtable<String, Conversacion>();
    ArrayList<Mensaje> mensajes=new ArrayList<Mensaje>();
    MensajeBBDD mbd=new MensajeBBDD(c);
    mbd.open();
    mensajes=(ArrayList<Mensaje>) mbd.getAllMensajes();
    mbd.close();
    int nMensajes=mensajes.size();
    Mensaje mAUX;
    String interlocutor;
    for(int i=0;i<nMensajes;i++){
        mAUX=mensajes.get(i);
        if(mAUX.getOrigen().equals(this.yo)){
            interlocutor=mAUX.getDestino();
        }else{
            interlocutor=mAUX.getOrigen();
        }
        if(conversaciones.containsKey(interlocutor)){
            conversaciones.get(interlocutor).setMensaje(mAUX);
        }else{
            conversaciones.put(interlocutor, new Conversacion());
            conversaciones.get(interlocutor).setMensaje(mAUX);
            Contacto contacto=Contactos.findContactoByNumero(interlocutor);
            if(contacto==null){
                contacto=new Contacto(interlocutor,interlocutor);
            }
            recientes.add(contacto);
        }
    }
}
```

Figura 85: cargar conversaciones almacenadas en la base de datos. CargadorConversaciones.java

La clase nos permite devolver una lista de Contactos, que podremos utilizar para construir la lista de conversaciones recientes gráficamente.

En el hilo que comprueba cada n segundos si tenemos mensajes pendientes en el servidor (Figura 86), también intenta enviar los mensajes almacenados en la tabla MensajesPendientes con el fin de enviarlos lo antes posible y eliminar sus registros de la base de datos.

```
private void lanzarMensajesPendientes() {
    GestorMensaje gm=new GestorMensaje(GUIPanelPrincipal.actualizador);
    ArrayList<Mensaje> pendientes=gm.getPendientes(this.getBaseContext());

    for(int i=0;i<pendientes.size();i++){
        Mensaje m=new Mensaje();
        m.setOrigen(pendientes.get(i).getOrigen());
        m.setDestino(pendientes.get(i).getDestino());
        m.setContenido(pendientes.get(i).getContenido());
        m.setID(pendientes.get(i).getID());
        m.setEnviado(pendientes.get(i).isEnviado());
        gm=new GestorMensaje(m);
        gm.enviar(this.getBaseContext(),2);
    }
}
```

Figura 86: Envío de mensajes pendientes. ReceptorMensajes.java

Planificación:

Requisitos del cliente:

Requisito	Pendiente	En Proceso	Completo
RF C1			X
RF C2			X
RF C3			X
RF C4			X
RF C5	X		
RF C6	X		
RF C7			X
RF C8			X

Requisitos del servidor:

Requisito	Pendiente	En Proceso	Completo
RF S1			X
RF S2			X
RF S3			X
RF S4			X

Actualmente se han cumplido los siguientes requisitos:

- Enviar / recibir mensajes del servidor
- Registro / Login
- Almacenar las conversaciones de forma persistente en el terminal, de forma que al salir del sistema, se queden almacenados los mensajes recibidos y enviados
- Gestionar una cola de mensajes pendientes de enviar en cuanto haya conexión

La funcionalidad básica de la aplicación está completa, en las próximas iteraciones se mejorará los aspectos gráficos y de usabilidad del sistema.

4.2.7. Ciclo 7: Mejora en la interfaz de conversación

Determinar Objetivos

En esta iteración los objetivos están marcados de forma muy clara. Se va a implementar una interfaz gráfica para interactuar con las conversaciones.

Hasta este momento las conversaciones se representaban como texto plano en una interfaz muy sencilla y funcional. Hasta ahora esta interfaz ha cumplido su cometido, pero una vez controlado el sistema de recepción y envío de mensajes, vamos a intentar que además de se haga correctamente, se muestre al usuario de forma profesional.

Resumiendo, los objetivos de este ciclo son:

- Diseño de una interfaz que muestre las conversaciones de forma visualmente atractiva a los usuarios
- Control de los mensajes pendientes de envío, mostrando al usuario una marca que identifique el mensaje como no enviado.

Para ilustrar el objetivo, nos basamos en cómo han implementado esta interfaz otras aplicaciones de la misma familia. Por ejemplo Whatsapp (Figura 87) y Line (Figura 88).



Figura 87: Ejemplo conversación WhatsApp



Figura 88: Ejemplo conversación Line

Análisis de riesgos

La interfaz gráfica de una aplicación es una de las partes más importantes a la hora de decantarse por una aplicación u otra. Según el artículo de Alberto la Calle [39], hasta el 45% del código de una aplicación está dedicado a la interfaz de usuario. No es de extrañar que en un entorno con tanta competencia como es el mercado de aplicaciones móviles se preste especial atención al diseño de una interfaz de usuario clara y eficiente. El diseño de una buena interfaz de usuario puede significar el éxito o el fracaso de una aplicación móvil.

Desarrollar, Verificar y Validar:

Como ya se expuso en la ilustración 12, las interfaces que implementan las actividades de Android están definidas en archivos XML dentro de la carpeta “Layout”.

Para diseñar la interfaz gráfica de las conversaciones de la aplicación, definimos el archivo “activity_conversacion.xml” y le damos la siguiente estructura (**¡Error! No se encuentra el origen de la referencia.**

```

1  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
2      android:layout_width="fill_parent"
3      android:layout_height="fill_parent">
4
5      <LinearLayout
6          android:id="@+id/linearLayout1"
7          android:layout_width="wrap_content"
8          android:layout_height="wrap_content"
9          android:layout_alignParentBottom="true"
10         android:layout_centerHorizontal="true"
11         android:gravity="center"
12         android:orientation="horizontal" >
13
14         <EditText
15             android:id="@+id/editText2"
16             android:layout_width="wrap_content"
17             android:layout_height="wrap_content"
18             android:ems="10" />
19
20         <Button
21             android:id="@+id/button1"
22             style="?android:attr/buttonStyleSmall"
23             android:layout_width="wrap_content"
24             android:layout_height="wrap_content"
25             android:text="enviar" />
26
27
28     </LinearLayout>
29
30     <ListView
31         android:id="@+id/list"
32         android:layout_width="match_parent"
33         android:layout_height="match_parent"
34         android:layout_above="@+id/linearLayout1"
35         android:layout_marginLeft="5dp"
36         android:layout_marginRight="5dp"
37         android:divider="#00000000"
38         android:dividerHeight="30dp" >
39
40     </ListView>
41
42 </RelativeLayout>

```

Figura 89: activity_conversacion.xml

Analizando el contenido del fichero comprobamos que está formado por un LinearLayout inferior que contiene una caja de texto donde escribiremos los mensajes y un botón enviar.

La interfaz también contiene un listview que abarcará todo el contenido de la pantalla excluyendo el linearLayout anterior, que contendrá la lista de todos los mensajes de esa conversación. El formato de cada elemento de ese listview está definido en otro fichero xml (**¡Error! No se encuentra el origen de la referencia.** 90)

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:tools="http://schemas.android.com/tools"
4      android:layout_width="wrap_content"
5      android:layout_height="wrap_content"
6      android:layout_marginLeft="50dp"
7      android:layout_marginRight="50dp">
8
9      <ImageView
10         android:id="@+id/bocadillo_entrada"
11         android:layout_width="20dp"
12         android:layout_height="20dp"
13         android:scaleType="fitXY"
14         android:layout_alignParentLeft="true"
15         android:layout_alignParentTop="true"
16         android:src="@drawable/entrante"
17         android:visibility="invisible" />
18
19
20      <ImageView
21         android:id="@+id/bocadillo_salida"
22         android:layout_width="20dp"
23         android:layout_height="20dp"
24         android:scaleType="fitXY"
25         android:layout_alignParentRight="true"
26         android:layout_alignParentTop="true"
27         android:src="@drawable/saliente_ok"
28         android:visibility="invisible" />
29
30
31      <TextView
32         android:id="@+id/texto_sobre_imagen"
33         android:layout_width="wrap_content"
34         android:layout_height="wrap_content"
35         android:textSize="20dp"
36         android:minHeight="20dp"
37         android:minWidth="150dp"
38         android:gravity="center"
39         android:layout_alignParentTop="false"
40         android:scaleType="fitXY"/>
41         <!--android:layout_toLeftOf="@+id/bocadillo_salida"-->
42         <!--android:background="@drawable/bocadillo_salida"-->
43
44
45
46
47  </RelativeLayout>

```

Figura 90: elemento_conversacion2.xml

Como se aprecia en la figura 90, cada elemento de conversación está formado por un TextView posicionado en el centro, y dos ImageView posicionados uno a la izquierda y otro a la derecha. En función de si el mensaje es de entrada o de salida solo será visible uno de estos dos.

Una vez definida la interfaz, le damos funcionalidad en tiempo de ejecución; El adaptador de listas “ConverAdapter.java” es el encargado de analizar las conversaciones y darle una apariencia al mensaje u otro (Figura 91) :

```
TextView texto = (TextView) rowView.findViewById(R.id.texto_sobre_imagen);
Mensaje item = this.items.get(position);
texto.setText(item.getContenido());
RelativeLayout.LayoutParams params = new RelativeLayout.LayoutParams(RelativeLayout.LayoutParams.MATCH_PARENT, RelativeLayout.LayoutParams.WRAP_CONTENT);
Imageview bocadilloEntrada=(Imageview) rowView.findViewById(R.id.bocadillo_entrada);
Imageview bocadilloSalida=(Imageview) rowView.findViewById(R.id.bocadillo_salida);

if(item.getOrigen().equals(me)){//mensaje de salida
    texto.setBackgroundResource(R.drawable.bocadillo_salida);
    Resources res = this.context.getResources();
    Drawable drawable=null;
    if(item.isEnviado()){

        drawable = res.getDrawable(R.drawable.saliente_ok);

    }else{
        drawable = res.getDrawable(R.drawable.saliente_temporal);
    }
    bocadilloSalida.setImageDrawable(drawable);
    params.addRule(RelativeLayout.LEFT_OF, bocadilloSalida.getId());
    texto.setLayoutParams(params);
    bocadilloSalida.setVisibility(View.VISIBLE);
    bocadilloEntrada.setVisibility(View.INVISIBLE);

}else{//mensaje de entrada
    texto.setBackgroundResource(R.drawable.bocadillo_entrada);
    params.addRule(RelativeLayout.RIGHT_OF, bocadilloEntrada.getId());
    texto.setLayoutParams(params);
    bocadilloEntrada.setVisibility(View.VISIBLE);
    bocadilloSalida.setVisibility(View.INVISIBLE);
}
```

Figura 91: converAdapter.java

Se analiza cada mensaje de la conversación y en función de si es de entrada o de salida, se activa un ImageView u otro del fichero “elemento_conversacion2.xml”.

En caso de que sea un mensaje saliente, también comprueba que si se ha enviado correctamente o si sigue en la cola de mensajes pendientes de envío.

Con estas modificaciones visuales, la interfaz del activity “GUIConversacion.java” quedaría (Figura 92); **Error! No se encuentra el origen de la referencia.:**



Figura 92: Ejemplo de conversación con la interfaz mejorada

Se han marcado con números ciertos elementos de la interfaz para explicar a continuación su procedencia:

1. Mensaje entrante
2. Mensaje saliente enviado correctamente
3. Mensaje saliente en cola de salida, esperando a que se restablezca la conexión con el servidor para ser enviado automáticamente.

Planificación

Requisitos del cliente:

Requisito	Pendiente	En Proceso	Completo
RF C1			X
RF C2			X
RF C3			X
RF C4			X
RF C5	X		
RF C6	X		
RF C7			X
RF C8			X

Requisitos del servidor

Requisito	Pendiente	En Proceso	Completo
RF S1			X
RF S2			X
RF S3			X
RF S4			X

En este ciclo no se ha implementado ningún requisito funcional pero se ha mejorado la apariencia de la aplicación, algo muy importante en las aplicaciones para dispositivos móviles.

Para cumplir con los objetivos planteados solo nos queda por cumplir dos requisitos: control del cierre de la aplicación e implementación de grupos de conversación. Este último será implementado cerca del cierre del proyecto. Pero el primer requisito será implementado en la próxima iteración junto a nuevas mejoras visuales.

4.2.8. Ciclo 8: Ciclo de vida del cliente y Action Bar

Determinar Objetivos

El ciclo actual, abarca dos objetivos muy distintos y que poco tienen que ver, pero que al ser a priori sencillos de llevar a cabo se han agrupado en el mismo ciclo.

El primero objetivo es controlar el cierre de la aplicación con el fin de recibir mensajes de forma constante hasta que el usuario de forma explícita cierre la aplicación.

Para llevar a cabo estas modificaciones se debe conocer el ciclo de vida de un Activity de Android. El ciclo de vida se ilustra de forma auto-explicativa en la Figura 93:

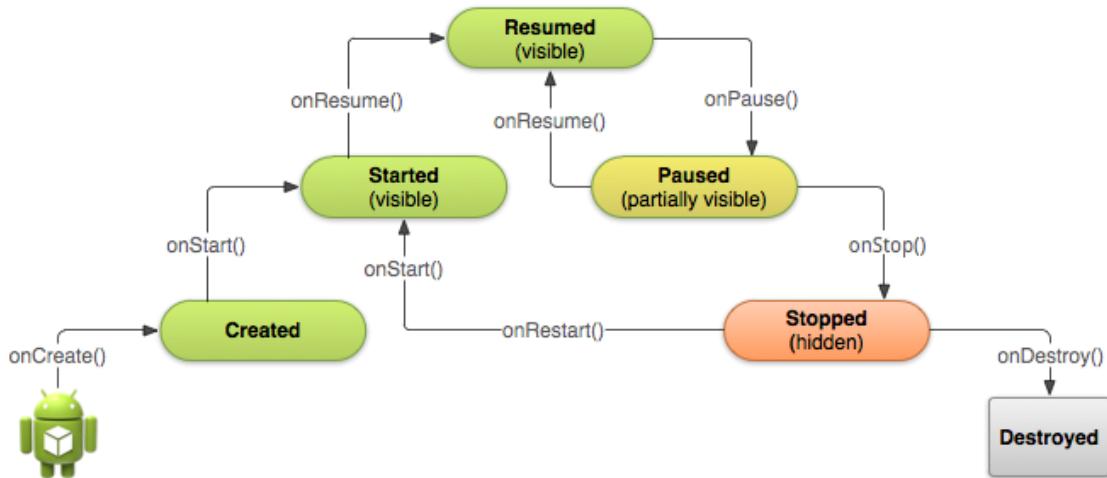


Figura 93: Ciclo de vida de Actividad Android. Fuente: <http://developer.android.com/training/basics/activity-lifecycle/start.html>

En principio, la aplicación sólo recibirá mensajes cuando se encuentre en un estado “Started” o “Resumed”, por lo que forzaremos a la aplicación a permanecer en estado “Started” ya que al pulsar el botón “atrás” de cualquier terminal la aplicación pasa automáticamente a “Stopped”.

El otro objetivo para este ciclo es mejorar la apariencia de la barra de título, incluyendo botones en ella. Es una forma de mostrar botones que se ha puesto de moda últimamente, ya que no ha tenido soporte nativo de Android hasta hace pocas versiones.

Análisis de riesgos

En cuanto al primer requisito, el del control del ciclo de vida de la aplicación, hay un riego bastante elevado, y es que al estar la aplicación en continua ejecución supone un uso más elevado de la batería del dispositivo. Tampoco se sabe a priori hasta qué punto el sistema operativo Android nos permitirá evitar el cierre de la aplicación, por lo que existe un riego bastante elevado de que este ciclo fracase.

Otro riego a tener en cuenta es la posibilidad de recibir mensajes con la aplicación ejecutándose en segundo plano, por lo que habrá que implementar algún sistema de notificaciones para informar al usuario. Una vez llegados a este punto no debe ser muy difícil encontrar documentación al respecto, por lo que implicaría un riego relativamente bajo.

Por otro lado, el requisito de implementar opciones en la barra de título, tiene un riesgo significativamente bajo, puesto que únicamente se trata de implementar botones para operaciones que ya están disponibles en otro sitio.

Desarrollar, Verificar y Validar:

En primer lugar se va a implementar un sistema para evitar que la aplicación se cierre. La idea es modificar el comportamiento de los botones de cierre y home para forzar a la aplicación a entrar en un modo estable y seguir ejecutándose en segundo plano. Con estable nos referimos a cerrar la conversación actual y volver al menú principal (GUIPanelPrincipal).

En la Figura 94 se sobrescribe el método onBackPressed() forzando a la aplicación a minimizarse y seguir su ejecución en un segundo plano al pulsar el botón Back” del terminal. Lo mismo se hace con el método onDestroy() que Android ejecuta automáticamente cuando desea cerrar la aplicación de forma completa.

```
@Override  
protected void onDestroy() {  
    moveTaskToBack(true);  
}  
  
//hacemos que en vez de salir se minimice y siga funcionando  
@Override  
public void onBackPressed() {  
    moveTaskToBack(true);  
}
```

Figura 94: Modificación del comportamiento de los botones de salida. GUIPanelPrincipal.java

Por si se desea realmente cerrar la aplicación de forma completa se ha implementado una opción “Salir” en el menú principal. Pulsar esta opción implica que pedimos explícitamente que se cierre la aplicación de forma completa, y dejar de recibir mensajes.

Para llevar a cabo este cierre se ha optado por realizar una llamada a una primitiva del sistema operativo que cierre todos los procesos relacionados con la actual aplicación (Figura 95).

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()) {
        {
            case R.id.action_new:
                lanzarGUISeleccionContacto();
                break;
            case R.id.action_search:
                Toast.makeText(this, "Search", Toast.LENGTH_SHORT).show();
                break;
            case R.id.action_Nuevo_Grupo:
                lanzarPanelNuevoGrupo();
                break;
            case R.id.action_salir:
                android.os.Process.killProcess(android.os.Process.myPid());
            default:
                return super.onOptionsItemSelected(item);
        }
        return true;
}
```

Figura 95: Primitiva al sistema para cerrar la aplicación

Con estas modificaciones, la aplicación será imposible de cerrar salvo que se lo pidamos explícitamente a través del menú principal. Actualmente el sistema recibe mensajes en segundo plano, y funciona como si estuviera en primer plano. Ahora se debe implementar un sistema para que el sistema nos notifique (sólo si la aplicación está en segundo plano) de que ha recibido nuevos mensajes.

Para notificar al usuario, primero debemos comprobar que la aplicación está en segundo plano. Para ello, dentro del runnable mUpdateResults que actualiza las conversaciones cuando llega un nuevo mensaje, se añade el siguiente fragmento de código (Figura 96):

```
if(!hasWindowFocus()){
    if(enConversacion){
        if(numeroEnConversacion!=c.getNumero()){
            lanzarNotificacion();
        }
    }else{
        lanzarNotificacion();
    }
}
```

Figura 96: Comprobación de estado de aplicación para notificar al usuario

Solo debemos lanzar una notificación al usuario en caso de que “!hasWindowsFocus” sea falso, y no estemos en ninguna conversación, lo que significaría que la aplicación está minimizada. Tambien se lanzará cuando “!hasWindowsFocus” sea verdadero pero estemos en una conversación distinta a la que pertenece el mensaje que acaba de llegar.

En definitiva, “!hasWindowsFocus” solo nos indica si GUIPanelPrincipal esta mostrándose, y no siempre que no esté mostrándose hay que lanzar la notificación al usuario.

La función “lanzarNotificación” (Figura 97) crea una notificación de usuario utilizando las clases NotificationManager y NotificationCompat. En esta función se puede modificar de forma bastante flexible el contenido de la notificación, texto, imagen, y el activity que se debe lanzar al pulsar sobre la notificación.

```

public void lanzarNotificacion(){
    long[] rafagasVibracion = {0, 100, 1000};
    Uri alarmSound = RingtoneManager.getDefaultUri(RingtoneManager.TYPE_NOTIFICATION);

    NotificationManager mNotificationManager =
        (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);

    NotificationCompat.Builder mBuilder =
        new NotificationCompat.Builder(this)
            .setSmallIcon(R.drawable.logo_blue)
            .setContentTitle("TalkMe")
            .setVibrate(rafagasVibracion)
            .setSound(alarmSound)
            .setLights(0xff00ff00, 300, 1000)
            .setContentText("Nuevo mensaje recibido");

    Intent notIntent = new Intent(this, GUIPanelPrincipal.class);
    PendingIntent contIntent = PendingIntent.getActivity(
        this, 0, notIntent, 0);

    mBuilder.setContentIntent(contIntent);

    mNotificationManager.notify(1, mBuilder.build());
}

```

Figura 97: lanzarNotificacion. GUIPanelPrincipal.java

Es importante borrar las notificaciones una vez que se ha accedido al sistema para ver su contenido. Esto se hace sobrescribiendo los métodos “onNewIntent”, “onResume” y “onRestart” (Figura 98), ya que son los puntos de entrada a la aplicación, y nos interesa que las notificaciones sean borradas una vez se acceda a este.

```

@Override
protected void onNewIntent(Intent intent) {
    super.onNewIntent(intent);
    mNotificationManager.cancelAll();
}

@Override
protected void onResume() {
    super.onResume();
    numeroEnConversacion="";
    enConversacion=false;
    mNotificationManager.cancelAll();

}

@Override
protected void onRestart() {
    super.onRestart();
    enConversacion=false;
    mNotificationManager.cancelAll();
}

```

Figura 98: Eliminación de notificaciones de usuario

Con estas modificaciones se ha cubierto completamente el primer objetivo del actual ciclo. Se ha controlado que la aplicación solo se cierre bajo petición expresa del usuario, y se ha creado un sistema de notificaciones al usuario que le avisan cuando la aplicación (trabajando en segundo plano) recibe mensajes de otros usuarios.

Para implementar el segundo requisito de este ciclo, hay que usar unas librerías externas de Android. Para ello se carga el proyecto “AppCompat” situado en “.../sdk/extras/Android/support/v7” y se añade al build path del proyecto. Para saber más de estas librerías externas consultar la documentación oficial de *Android ActionBarActivity* [40].

Con esta librería cargada en el proyecto se debe realizar un pequeño cambio en cada Activity del proyecto que se desee que incluya una barra de título con botones (ActionBarActivity).

El cambio a realizar (Figura 99) consiste en que el Activity ya no herede de la clase Activity, sino de ActionBarActivity

```
public class GUIPanelPrincipal extends ActionBarActivity{
```

Figura 99: Implementación de ActionBarActivity

Con esto el Activity GUIPanelPrincipal está preparado para incluir botones en su barra de título. Para añadirlos se debe modificar el fichero de menú de este Activity, almacenado en la carpeta “menu” de “res” (Figura 100)

```

1  <menu xmlns:android="http://schemas.android.com/apk/res/android"
2      xmlns:sgoliver="http://schemas.android.com/apk/res-auto" >
3
4      <item
5          android:id="@+id/action_search"
6          android:orderInCategory="100"
7          sgoliver:showAsAction="ifRoom"
8          android:icon="@drawable/abc_ic_search"
9          android:title="@string/action_search"/>
10
11     <item
12         android:id="@+id/action_new"
13         android:orderInCategory="100"
14         sgoliver:showAsAction="ifRoom"
15         android:icon="@drawable/ic_menu_invite"
16         android:title="@string/action_settings"/>
17
18     <item
19         android:id="@+id/action_new"
20         android:orderInCategory="100"
21         sgoliver:showAsAction="never"
22         android:title="Nueva Conversacion"/>
23
24     <item
25         android:id="@+id/action_Nuevo_Grupo"
26         android:orderInCategory="100"
27         sgoliver:showAsAction="never"
28         android:title="Nuevo Grupo"/>
29
30     <item
31         android:id="@+id/action_salir"
32         android:orderInCategory="100"
33         sgoliver:showAsAction="never"
34         android:title="Salir"/>
35
36
37
38 </menu>

```

Figura 100: menu/guipanelprincipal.xml

En este fichero se definen las diferentes opciones del menú de opciones de la aplicación (Menú accesible siempre pulsando el botón menú del terminal). Se pueden distinguir dos tipos de elementos: los del menú inferior, y los del menú de la barra de título. Para definirlos en uno u otro lugar se utiliza la *opción sgoliver:showAsAction= “never”* para definirlos en el menú inferior o *sgoliver:showAsAction= “ifRoom”* para definirlos en la barra de título. A todos los elementos los identifica un id, que será recogido por la aplicación al ser pulsada la opción, para poder actuar en consecuencia (Figura 101).

```

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()) {
        case R.id.action_new:
            lanzarGUISelecciónContacto();
            break;
        case R.id.action_search:
            Toast.makeText(this, "Search", Toast.LENGTH_SHORT).show();
            break;
        case R.id.action_Nuevo_Grupo:
            lanzarPanelNuevoGrupo();
            break;
        case R.id.action_salir:
            if(conectado){
                //this.serverThread.stop();
            }
            android.os.Process.killProcess(android.os.Process.myPid());
        default:
            return super.onOptionsItemSelected(item);
    }
    return true;
}

```

Figura 101: Selección de opción del menú

En la figura 102 se muestra cómo queda la interfaz gráfica del Activity GUIPanelPrincipal. Los botones del ActionBar tienen la misma funcionalidad que los del menú inferior. Se ha añadido un botón de búsqueda aún sin funcionalidad añadida.



Figura 102: Estado GUIPanelPrincipal. Ciclo 8

Si se desea implementar botones en el ActionBar en otros Activities el procedimiento sería similar al seguido en esta interfaz y se resume como:

1. Extender el Activity de ActionBarActivity
2. Modificar el fichero xml del menú correspondiente al Activity a modificar
3. Marcar al elemento del menú con la propiedad *sgoliver:showAcAction=“ifRoom”*

Planificación

Requisitos del cliente:

Requisito	Pendiente	En Proceso	Completo
RF C1			X
RF C2			X
RF C3			X
RF C4			X
RF C5			X
RF C6	X		
RF C7			X
RF C8			X

Requisitos del servidor

Requisito	Pendiente	En Proceso	Completo
RF S1			X
RF S2			X
RF S3			X
RF S4			X

Llegados a este punto, tenemos una aplicación completamente funcional y se han cumplido los requisitos establecidos en [3.2] a excepción de las conversaciones de grupo.

- La aplicación envía y recibe mensajes de otros usuarios
- La aplicación gestiona y almacena correctamente los mensajes
- Se ha establecido una gestión de notificaciones de usuario de forma satisfactoria

En la aplicación servidor, también se han cumplido todos los requisitos planteados (desde el ciclo 5).

No obstante hay un sistema que sigue sin convencer al autor y director del presente proyecto: La recepción de mensajes. Pese a que en principio funciona correctamente, el sistema implementado en el ciclo 5 no parece el más eficiente, así que tras unas reuniones con el director del proyecto se ha acordado, que llegados a este punto del proyecto, se dedique un ciclo exclusivamente a la mejora de este sistema. Así pues el siguiente ciclo tratará de buscar un sistema alternativo, posiblemente mucho más eficiente al problema de la recepción de mensajes planteado en ciclo 5.

4.2.9. Ciclo 9: Cambio en el sistema de recepción de mensajes

Determinar Objetivos

En este ciclo, como se comentó en la sección 4.2.8, se va a tratar de implementar una alternativa al actual sistema de recepción de mensajes, que pese a su correcto funcionamiento, no convence al autor del proyecto, ni al director de este.

Se recuerda, que como se expone en la sección 4.2.5, el actual sistema de recepción de mensajes consta de un proceso trabajando en segundo plano que, cada X segundos, realiza una petición HTTP al servidor de la aplicación, y este responde a la petición con los mensajes almacenados a su nombre que previamente han sido enviados por otros usuarios. Posteriormente, una vez recibidos por el destinatario del mensaje, son eliminados de la base de datos del servidor.

Las razones por las que este sistema no parece el más eficiente posible son las siguientes:

- Elevado consumo de batería por parte del cliente, ya que necesita que siempre haya un proceso en segundo plano que cada pocos segundos ejecute una petición HTTP.
- Elevado consumo de recursos por parte del cliente por las mismas razones citadas anteriormente.
- Elevado consumo de recursos por parte del servidor que tiene que atender una petición cada pocos segundos por cada cliente conectado a él. Además la mayoría de estas peticiones serían inútiles ya que la mayor parte del tiempo la aplicación no tiene mensajes entrantes. En principio puede atenderlas correctamente, pero si llegara el momento en que esta aplicación tuviera miles de usuarios conectados, este sistema no aguantaría la carga de peticiones por lo que es un sistema poco escalable
- Sobrecarga de la base de datos, ya que para cada mensaje se realizarían al menos dos accesos a esta. Uno para almacenar el mensaje entrante, y otro para recuperarlo, una vez atendida la petición de mensajes del cliente receptor de este.
- No se envían los mensajes en tiempo real, ya que el receptor debería esperar a que su proceso de recepción de mensajes realice la petición al servidor, y esto podría alargarse varios segundos.

Por estas razones, y sobre todo la falta de escalabilidad se ha optado por buscar un sistema alternativo a la recepción de mensajes.

Tras varias reuniones con el director del proyecto, y tras el descarte de otras soluciones como activar un puerto de escucha en cada cliente, que además de no solucionar todos los problemas explicados anteriormente, añadía otros cuantos, se optó por la solución más elegante y utilizada: Notificaciones PUSH.

Las notificaciones PUSH es un moderno mecanismo de comunicación Cliente-Servidor en que el servidor es quien inicia la comunicación.

Se quiere justificar la ausencia de este sistema en ciclos anteriores con la falta de experiencia del alumno en desarrollo de aplicaciones móviles, dado que este sistema está pensado exclusivamente para ellas, aunque se pueda utilizar en otros ámbitos.

Tras la pertinente investigación y reuniones con el director del proyecto se decidió implementar este sistema para la recepción de mensajes a través de la aplicación de *Google GCM (Google Cloud Messaging)* [41]

El sistema GCM consiste en tener a los clientes registrados en un servidor de Google asociado a la aplicación, y el servidor de la aplicación conoce el identificador de cada usuario (porque previamente se lo ha suministrado). Entonces cuando el servidor recibe un mensaje, analiza el destino, y se lo envía al servidor GCM marcándole como destino el destino del mensaje. El servidor GCM será el que se encargue de enviar el mensaje al destinatario correspondiente. El cliente de la aplicación solo tendrá que implementar el código necesario para recibir mensajes del servidor GCM.

El proceso, de forma general, se ilustra en la figura 103:

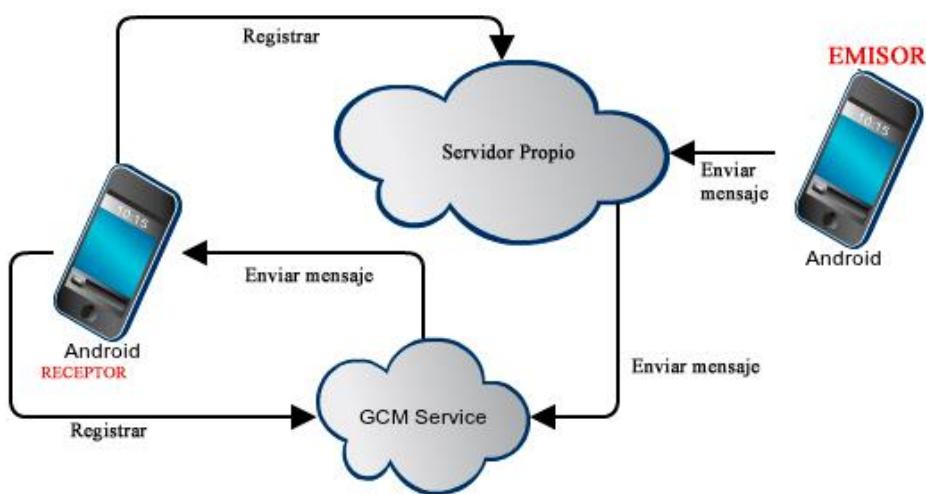


Figura 103: Proceso de envío de mensajes por GCM

El sistema GCM se adapta perfectamente a las necesidades del proyecto ya que:

- No será necesario mantener una comunicación cliente-servidor, ahorrando en recursos
- No será necesario almacenar los mensajes en la base de datos del servidor, ya que tan pronto le lleguen al servidor serán enviados al servidor GCM
- La comunicación será en tiempo real por las mismas razones citadas anteriormente.

Análisis de riesgos

El sistema a desarrollar en este ciclo cambia completamente la arquitectura del sistema implementado, por lo que habrá que realizar cambios bastante profundos en el sistema actual para adaptarlo a este sistema de recepción de mensajes. Todos estos cambios se realizarán sin la certeza de que este nuevo sistema consiga ser más eficiente que el anterior, además que implicará una inversión en tiempo que no estaba planificada, y, la entrega del proyecto y su éxito corre bastante peligro. Por lo cual estamos ante un riego bastante elevado. Es el ciclo de desarrollo en el que más riegos corremos, ya que podría influir negativamente en todo lo construido anteriormente, debido a su naturaleza completamente distinta.

Pese a ello, tras analizar la situación se decide la implementación del sistema, ya que en caso de éxito, las ventajas son muchas, ya que además de cubrir las deficiencias del sistema anterior, convertirá a la aplicación actual en una mucho más escalable.

Desarrollar, Verificar y Validar

Antes de comenzar con la exposición del desarrollo de este ciclo se ha de comentar que nos hemos basado en la documentación que Android nos ofrece para implementar este tipo de servicios GCM [41].

En primer lugar se expone los cambios en el cliente del sistema.

El primer cambio ha sido eliminar la clase ReceptorMensajes.java que implementaba un servicio en segundo plano que realizaba peticiones de mensajes al servidor de la aplicación. En su lugar se ha implementado el sistema descrito a continuación.

Para empezar, el sistema nada mas loguearse en el servidor de la aplicación, comprueba si está registrado en el servidor GCM, y en caso contrario se registra utilizando como identificador el número de teléfono con el que se registró en el servidor de la aplicación (Figura 104)

```

private void conectarse() {

    context = getApplicationContext();
    Toast.makeText(getApplicationContext(),"empezamos", Toast.LENGTH_SHORT).show();
    gcm = GoogleCloudMessaging.getInstance(GUIPanelPrincipal.this);
    Toast.makeText(getApplicationContext(),"recupera info", Toast.LENGTH_SHORT).show();

    //Obtenemos el Registration ID guardado
    regid = getRegistrationId(context);

    //Si no disponemos de Registration ID comenzamos el registro
    if (regid.equals("")) {
        Toast.makeText(getApplicationContext(),"no esta registrado", Toast.LENGTH_SHORT).show();
        TareaRegistroGCM tarea = new TareaRegistroGCM();
        tarea.execute("");
        Toast.makeText(getApplicationContext(),"registro completado", Toast.LENGTH_SHORT).show();
    }else{
        Toast.makeText(getApplicationContext(),"ya esta registrado", Toast.LENGTH_SHORT).show();
    }
}

```

Figura 104: Comprobación de credenciales GCM. GUIPanelprincipal.java

En caso de no estar registrado, como sería la normal en la primera ejecución, el sistema registrará el dispositivo en el servidor GCM utilizando para ello la clase TareaRegistroGCM (Figura 105) ejecutada en un segundo plano para no congelar la interfaz gráfica del sistema.

```

private class TareaRegistroGCM extends AsyncTask<String, Integer, String>
{
    @Override
    protected String doInBackground(String... params)
    {
        String msg = "";

        try
        {
            if (gcm == null)
            {
                gcm = GoogleCloudMessaging.getInstance(context);
            }

            //Nos registramos en los servidores de GCM
            regid = gcm.register(SENDER_ID);

            Log.d(TAG, "Registrado en GCM: registration_id=" + regid);
            //Nos registramos en nuestro servidor
            boolean registrado = registroServidor(user, pass, regid);
            //Guardamos los datos del registro
            if(registrado)
            {
                setRegistrationId(context, params[0], regid);
            }
        }
        catch (IOException ex)
        {
            Log.d(TAG, "Error registro en GCM:" + ex.getMessage());
        }

        return msg;
    }
}

```

Figura 105: TareaRegistroGCM. Registro en servidor GCM

Una vez registrado en el servidor GCM y habiendo obtenido el ID único que este servidor nos asocia, el siguiente paso es indicárselo a nuestro propio servidor (Figura 106), para que este sea capaz de enviarle mensajes al servidor GCM indicándole como destino nuestro ID.

```
private boolean registroServidor(String usuario, String pass , String regId)
{
    Toast toast1;

    context = getApplicationContext();
    GestorCredencial credenciales=new GestorCredencial(usuario,pass,regId);
    boolean res=credenciales.enviar();
    return res;
}
```

Figura 106: registroServidor. GUIPanelPrincial.java

El siguiente paso, obviando las operaciones del servidor que se expondrán mas adelante, es implementar un sistema que trabajando en segundo plano reciba los mensajes del servidor GCM y los gestione de la misma forma que sistema anterior.

Para ello vamos a utilizar un BroadcastReceiver que filtre los mensajes GCM (Figura 107), y se apoye en un Service (Figura 108) que se encargue de gestionar cada mensaje de manera independiente y en segundo plano

```
public class GCMBroadcastReceiver extends WakefulBroadcastReceiver
{
    @Override
    public void onReceive(Context context, Intent intent)
    {
        ComponentName comp =
            new ComponentName(context.getPackageName(),
                GCMIntentService.class.getName());

        startWakefulService(context, (intent.setComponent(comp)));
        setResultCode(Activity.RESULT_OK);
    }
}
```

Figura 107: GCMBroadcastReceiver

```

@Override
protected void onHandleIntent(Intent intent)
{
    GoogleCloudMessaging gcm = GoogleCloudMessaging.getInstance(this);

    String messageType = gcm.getMessageType(intent);
    Bundle extras = intent.getExtras();

    if (!extras.isEmpty())
    {
        if (GoogleCloudMessaging.MESSAGE_TYPE_MESSAGE.equals(messageType))
        {

            gestionarMensaje(extras);
            //mostrarNotification(extras);
        }
    }

    GCMBroadcastReceiver.completeWakefulIntent(intent);
}

```

Figura 108: GCMIntentService

Una vez llegamos a ejecutar la función “gestionarMensaje” (Figura 108), el mensaje habrá sido recibido, y el proceso de gestión del mensaje es similar al implementado con anterioridad.

Con estos últimos cambios el cliente de la aplicación está preparado para recibir mensajes GCM y gestionarlos de forma eficiente. A continuación se describe los cambios producidos en el servidor a favor de este nuevo sistema de envío/recepción de mensajes.

En primer lugar se ha eliminado la tabla “mensajes” de la base de datos, así como todo el código relacionado con el almacenamiento y recuperación de información de esta tabla, ya que con este sistema, los mensajes no se almacenarán jamás, sino que solo pasarán por el servidor como intermediario entre el usuario emisor y el servidor GCM.

El siguiente cambio también se produce en la base de datos, ya que la tabla usuarios debe almacenar ahora, además del usuario y la contraseña, el identificador de GCM que recibe del cliente una vez que se registra.

Después, se ha modificado la página *recibir.jsp*. Como se aprecia en la línea 18 de la figura 109, se ha modificado el procedimiento, y en lugar de almacenar el mensaje, se utiliza la función enviar.

```

13     String texto=null;
14     try{
15         texto=request.getParameter("mensaje");
16         System.out.println(request.getRemoteHost());
17         GestorMensaje gm=new GestorMensaje(texto);
18         gm.enviar();
19         gm.imprimirPorPantalla();
20         long suma = gm.getSuma();
21         %>
22         <%=suma%>
23     <%
24     %>

```

Figura 109: recibir.jsp. Ciclo 9

La función enviar, es un intermediario que prepara el mensaje para ser enviado al servidor GSM mediante la clase GCMBroadcast (Figura 110).

- La variable estática SENDER_ID (línea 30) es un código que el servidor GCM nos suministra al registrarnos para identificar de forma única la aplicación
- La variable estática DROID_BIONIC (línea 31) es el identificador del dispositivo cliente, previamente rescatado de la base de datos, hacia el que va dirigido el mensaje.
- Los tipos de datos de las líneas 59, 60 y 61 serán necesarios para recuperar el mensaje en el cliente de la aplicación.
- El parámetro “tipo” de la línea 59 indica el tipo de mensaje que es, con vistas a, en futuros ciclos poder necesitar otro tipo de mensajes con otros parámetros diferentes.

```

26 public class GCMBroadcast {
27
28     private Mensaje m;
29     private static final Long serialVersionUID = 1L;
30     private static final String SENDER_ID = "AIzaSyB8OKI29qRf46hLWdlsQAU TJqN-fDRnESK";
31     private String DROID_BIONIC = "";
32     private List<String> androidTargets = new ArrayList<String>();
33
34     public GCMBroadcast(String idDestino, Mensaje m) {
35
36         this.DROID_BIONIC=idDestino;
37         this.m=m;
38         androidTargets.add(DROID_BIONIC);
39
40     }
41
42     public void doPost() throws ServletException, IOException {
43
44         try {
45             } catch (Exception e) {
46
47                 e.printStackTrace();
48                 return;
49             }
50             System.out.println("enviamos tipo 1");
51
52             Sender sender = new Sender(SENDER_ID);
53
54             Message message = new Message.Builder()
55
56                 .collapseKey("")
57                 .timeToLive(30)
58                 .delayWhileIdle(true)
59                 .addData("tipo", "1")//tipo 1= mensaje normal
60                 .addData("origen", this.m.getOrigen())
61                 .addData("contenido", this.m.getContenido())
62                 .build();
63
64             try {
65                 MulticastResult result = sender.send(message, androidTargets, 1);
66
67                 if (result.getResults() != null) {
68                     int canonicalRegId = result.getCanonicalIds();
69                     if (canonicalRegId != 0) {
70
71                         }
72                     } else {
73                         int error = result.getFailure();
74                         System.out.println("Broadcast failure: " + error);
75                     }
76                     System.out.println("gcm enviado");
77                 } catch (Exception e) {
78                     System.out.println("error");
79                     e.printStackTrace();
80                 }
81             }
82

```

Figura 110: GCMBroadcast. Servidor

Planificación:

Requisitos del cliente:

Requisito	Pendiente	En Proceso	Completo
RF C1			X
RF C2			X
RF C3			X
RF C4			X
RF C5			X
RF C6	X		
RF C7			X
RF C8			X

Requisitos del servidor:

Requisito	Pendiente	En Proceso	Completo
RF S1			X
RF S2			X
RF S3			X
RF S4			X

Con las modificaciones de este ciclo, pese a la inversión en tiempo y puesta en peligro del proyecto, la aplicación ha ganado en robustez, escalabilidad y eficiencia. Todo el sistema en general funciona bastante mejor.

Analizando los requisitos que faltan se ha tomado la decisión de implementar en el siguiente ciclo el único requisito funcional pendiente, RF C6, que consiste en implementar un sistema de conversaciones en grupo. Este siguiente ciclo, debería ser a priori el último ciclo de desarrollo del proyecto.

4.2.10. Ciclo 10: Conversaciones en Grupo

Determinar Objetivos

En el que es presumiblemente el último ciclo de desarrollo del TFG proyecto se va a desarrollar un sistema de conversaciones grupales, en el que participen 3 o más interlocutores al mismo tiempo.

Para ello un usuario deberá dar de alta un grupo, seleccionando varios usuarios de su lista de contactos. Él deberá iniciar la conversación, hasta que él no envíe un primer mensaje los demás interlocutores no tendrán constancia de la creación del grupo de conversación. Una vez se produzca este primer mensaje, cualquier interlocutor podrá enviar mensajes al grupo, siendo los receptores todos los participantes de dicho grupo.

Los pasos que se plantean para conseguir el objetivo de este ciclo son, a gran escala, los siguientes:

1. Creación, en el cliente, de una tabla grupo donde se almacena la información de cada grupo, además de una tabla “usuarios_grupo” y otra “mensajes_grupo” que relacione los usuarios y los mensajes respectivamente con el grupo. En el servidor también habría que implementar estas tablas excepto la de mensajes_grupo.
2. Interfaz de conversación distinta a GUIConversacion, en este caso se llamará GUIGrupo
3. Creación de una interfaz para la creación de grupos, que se llamará “GUI_Nuevo_Grupo”
4. Modificación del sistema de recepción de mensajes para analizar si es un mensaje normal, o un mensaje de grupo.

Análisis de riesgos

En una etapa tan avanzada del proyecto como esta, un cambio tan significativo como la introducción de conversaciones de grupo, podría poner en peligro el buen funcionamiento de las otras funcionalidades del sistema si no se realiza correctamente.

Se podría haber implementado esta funcionalidad como complemento de las otras, intentando cohesionar las funciones de una conversación normal con una de grupo, de hecho, la clase Grupo se podría haber implementado como una subclase de Conversación entre otras cosas. Sin embargo este tipo de cohesión, en una etapa tan avanzada del proyecto podría poner en peligro el buen funcionamiento del sistema, y con el fin de reducir los riesgos al mínimo, se ha decidido implementar todo el sistema

de conversaciones en grupo de forma paralela a las conversaciones normales, intentando que las conversaciones en grupo utilicen lo mínimo posible el código actual del proyecto, reduciendo notablemente los riesgos a la hora de desarrollar nuevas funciones.

Desarrollar, Verificar y Validar:

En primer lugar, se define el contenido de un Grupo. Para ello creamos una clase de conocimiento llamada Grupo, que contiene una lista de mensajes. En la figura 111 se ilustra los atributos que tiene un grupo y cómo se relaciona con los mensajes.

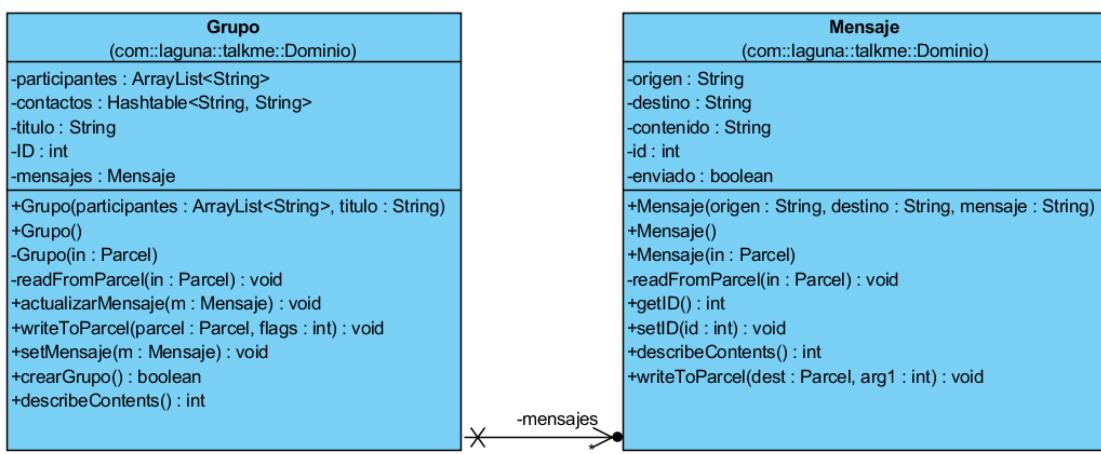


Figura 111: UML Grupo

Como muestra la figura 111, un grupo está compuesto de:

- Participantes: es una lista con los números de teléfono de todos los miembros del grupo
- Contactos: es una tabla que relaciona los números de teléfono de los participantes en el grupo, con el contacto de la agenda del usuario (si lo tiene en su agenda)
- Título: es el nombre que se le da al grupo de conversación
- ID: es un identificador único que tiene el grupo para diferenciarse de todos los demás en el servidor
- Mensajes: es el conjunto de mensajes que se ha enviado en el grupo. Se apoya en la clase Mensaje, como una conversación normal, pero en el destino del mensaje siempre está indicado el identificador del grupo.

También puede apreciarse que la clase Grupo implementa las funciones de la superclase Parcelable (`readFromParcel` y `writeToParcel`) para poder ser enviado de un Activity a otro.

Se crea un activity “GUI_Nuevo_Grupo” que nos muestra una interfaz (Figura 112) que nos permite crear un nuevo grupo de conversación

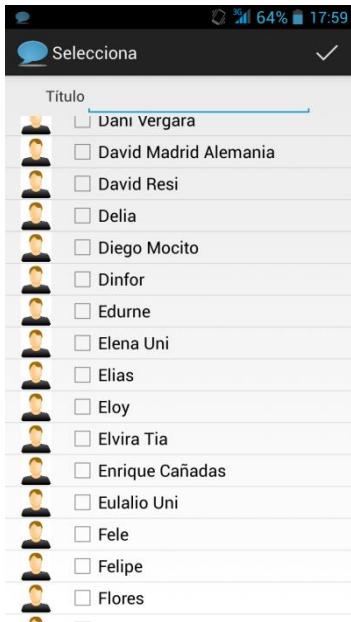


Figura 112:GUI_Nuevo_Grupo. Interfaz

Una vez se rellena el formulario con los datos del grupo, se crea una petición de creación de grupo. Para ello el proceso es el siguiente:

1. GUI_Nuevo_Grupo crea un objeto de la clase Grupo con los datos introducidos en el formulario y utiliza su función crearGrupo() (Figura 113):

```
private void enviarPeticionCreacionGrupo(ArrayList<String> numerosSeleccionados, String titulo) {  
    Grupo g=new Grupo(numerosSeleccionados, titulo);  
    boolean creado=g.crearGrupo();  
  
    if(creado){  
        Intent i = new Intent( this, GUI_Nuevo_Grupo.class );  
        setResult( Activity.RESULT_OK, i );  
        this.finish();  
    }else{  
        Toast toast1;  
        toast1=Toast.makeText(getApplicationContext(), "Error al crear el grupo ", Toast.LENGTH_LONG);  
        toast1.show();  
    }  
}
```

Figura 113: enviarPeticionCreacionGrupo. GUI_Nuevo_Grupo.java

2. La clase Grupo, a través de su función crearGrupo(), prepara un objeto de la clase CrearGrupo que trabaja en segundo plano, y le proporciona las propiedades del grupo (Figura 114)

```

public boolean crearGrupo() {
    CrearGrupo cg=new CrearGrupo();
    cg.setParametros(this.titulo,this.participantes);
    cg.execute();
    try {
        if(cg.get()){
            return true;
        }else{
            return false;
        }
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (ExecutionException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    return false;
}

```

Figura 114: crearGrupo. Grupo.java

3. La clase CrearGrupo envía una petición HTTP al servidor de la aplicación indicándole los datos del grupo que se desea crear (Figura 115).

```

public void enviar() {
    HttpClient client = new DefaultHttpClient();
    HttpPost post = new HttpPost(Rutas.getRuta("server") + Rutas.getRuta("crearGrupo"));
    List<NameValuePair> urlParameters = new ArrayList<NameValuePair>();
    urlParameters.add(new BasicNameValuePair("titulo", this.titulo));
    JSONArray jsArray = new JSONArray(participantes);
    urlParameters.add(new BasicNameValuePair("participantes", jsArray.toString()));
    urlParameters.add(new BasicNameValuePair("user", GUIPanelPrincipal.user));
    try {
        post.setEntity(new UrlEncodedFormEntity(urlParameters));
    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
    }
    try {
        HttpResponse respuesta=client.execute(post);

        HttpEntity entity = respuesta.getEntity();
        InputStream is = entity.getContent();
        this.respuesta= limpiarRespuesta(is);

    } catch (ClientProtocolException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch(Exception e){
        e.printStackTrace();
    }
}

```

Figura 115: enviar. CrearGrupo.java

El siguiente paso, se debe dar en el servidor. Antes de nada, se detallan los cambios en la base de datos para dar cabida a los grupos de conversación. En la Figura 116, se muestra en un diagrama Entidad-Relación que se ha añadido una entidad Grupo con clave principal ID auto-incrementable. Esta entidad Grupo tiene una relación muchos-a-muchos con la entidad ya existente usuario. El resultado en una tabla intermedia que relaciona en cada entrada una entidad grupo con una entidad usuario.

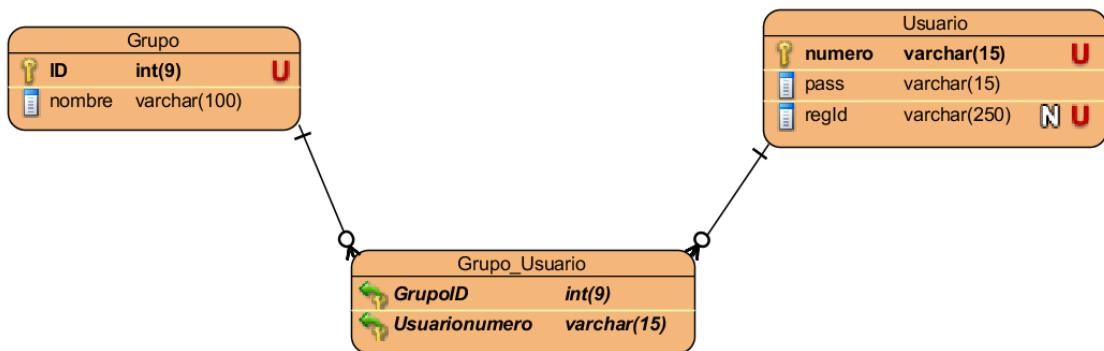


Figura 116: Base de datos del Servidor. Ciclo 10

Se recuerda que el almacén de mensajes no será necesario ya que el servidor únicamente se encargará de reenviarlos al servidor GCM (Ciclo 9).

Una vez actualizada la base de datos, se detalla el proceso de creación de un grupo de conversación desde que la petición llega al servidor de la aplicación.

La página crearGrupo.jsp es la encargada de recoger la petición de creación de grupo y desencadenar las operaciones pertinentes. El proceso que se realiza (Figura 117) se resume en los siguientes pasos:

1. Se reconstruye el objeto de tipo grupo, enviado mediante la petición HTTP.
2. Se almacena en la base de datos
3. Se notifica al creador del grupo de que el grupo se ha creado correctamente. Los demás participantes del grupo solo tendrán constancia de la creación del grupo una vez el creador de este haya enviado el primer mensaje.

```

<% String titulo=null;
String participantes=null;
String creador = null;
try{
    System.out.println("Recibiendo petición de crear grupo");
    titulo=request.getParameter("titulo");
    participantes=request.getParameter("participantes");
    creador=request.getParameter("user");
    System.out.println(participantes);
    Grupo g=new Grupo(titulo,participantes);
    g.almacenarEnBBDD();
    g.enviarNotificacion(creador);
    System.out.println("Grupo creado");

    %>ok

<%

```

Figura 117: crearGrupo.jsp. Recepción de petición de creación de grupo

Los pasos 1 y 2 se realizan de la misma manera que se ha estado haciendo la recepción de mensajes, o credenciales de usuario. Sin embargo se considera importante detallar el tercer paso de esta operación.

Para notificar al usuario creador del grupo que el grupo ha sido creado correctamente, se ha optado por hacerlo mediante una notificación POST, a través del servidor GCM (Figuras 118 y 119). Esta notificación incluirá los datos del grupo, que el usuario reconstruirá (Figura 120) y almacenará en su base de datos (Figura 121).

```

public boolean enviarNotificacion(String user){
    UsuarioBBDD ubd=new UsuarioBBDD();
    String idDestino="";
    if(ubd.existeNumEnBaseDeDatos(user)){
        idDestino=ubd.getID(user);
        GCMConfirmacionGrupo envio=new GCMConfirmacionGrupo(idDestino,codificarGrupo());
        try {
            envio.doPost();
            System.out.println("enviado correctamente XD");
        } catch (ServletException e) {
            // TODO Auto-generated catch block
            System.out.println("no enviado!");
        } catch (IOException e) {
            System.out.println("error!");
            e.printStackTrace();
        }
    }
    return false;
}

```

Figura 118: enviarNotificacion. Grupo.java (Servidor)

```

public GCMConfirmacionGrupo(String idDestino, String grupo) {
    this.DROID_BIONIC=idDestino;
    this.grupo=grupo;
    androidTargets.add(DROID_BIONIC);
}

public void doPost() throws ServletException, IOException {
    try {
    } catch (Exception e) {
        e.printStackTrace();
        return;
    }
    Sender sender = new Sender(SENDER_ID);
    Message message = new Message.Builder()

    .collapseKey("")
    .timeToLive(30)
    .delayWhileIdle(true)
    .addData("tipo", "2")//tipo 2= confirmacion grupo
    .addData("grupo", grupo)
    .build();

    try {
        MulticastResult result = sender.send(message, androidTargets, 1);

        if (result.getResults() != null) {
            int canonicalRegId = result.getCanonicalIds();
            if (canonicalRegId != 0) {

            }
        } else {
            int error = result.getFailure();
            System.out.println("Broadcast failure: " + error);
        }
        System.out.println("gcm enviado");
    } catch (Exception e) {
        System.out.println("error");
        e.printStackTrace();
    }
}

```

Figura 119: Confirmar la creación del grupo al creador. GCMConfirmacionGrupo.java

```

private void gestionarMensaje(Bundle extras) {
    int tipo=Integer.parseInt(extras.getString("tipo"));

    switch (tipo){
    case 1:
        leerMensajeNormal(extras);
        break;
    case 2://confirmacion grupo creado
        leerMensajeConfirmacionGrupo(extras);
        break;
    }
}

```

Figura 120: Filtro de mensajes en el cliente. GCMIntentService.java (Cliente)

```

private void almacenarGrupo(Grupo g) {
    GUIPanelPrincipal.actualizador.actualizar(g);
    GrupoBBDD mbd=new GrupoBBDD(GUIPanelPrincipal.contexto);
    mbd.open();
    mbd.insertarGrupo(g);
    mbd.close();
}

```

Figura 121: almacenarGrupo. GCMIntentService.java

Para actualizar la interfaz de usuario con el nuevo grupo, se ha recurrido al polimorfismo de Java. Implementamos otra función “actualizar” en el objeto Actualizador (Figura 122), pero al contrario que la función ya existente, esta no trae un mensaje nuevo, sino un grupo nuevo. Esta nueva función actualizará la interfaz de usuario del mismo modo que lo hace la otra con un nuevo mensaje.

```

public void actualizar(Grupo g) {
    gruposEnCache.put(Integer.toString(g.getID()), g);
    this.g=g;
    this.tipo=2;

    for(int i=0;i<this.mHandlerList.size();i++){
        this.mHandlerList.get(i).post(this.mUpdateResultsList.get(i));
    }
}

```

Figura 122: Crear un nuevo grupo ya confirmado con la función actualizar. Actualizador.java

En cuanto a la base de datos del cliente, también hay que realizar unas modificaciones muy parecidas a las del servidor, con la única diferencia de que en el cliente sí hay que almacenar los mensajes, lo que añade una nueva entidad a la relación usuarios-grupos. Para ilustrar esta relación de forma más estándar se recurre al siguiente diagrama Entidad-Relación (Figura 123)

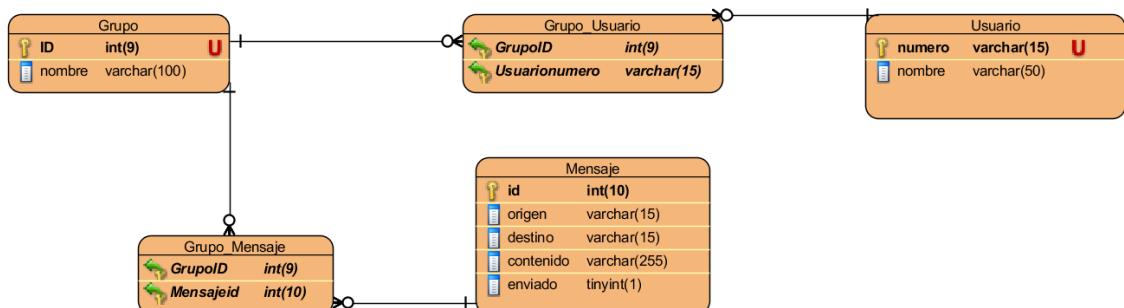


Figura 123: Diagrama Entidad-Relación Cliente. Implementación de grupos

El resultado de implementar el anterior diagrama en la base de datos relacional es, además de crear una tabla Grupo, la creación de dos tablas intermedias: “grupo_usuario” y “grupo_mensaje” que relacione los grupos con sus participantes y mensajes respectivamente.

El siguiente paso es diseñar un Activity que muestre las conversaciones de grupo y nos permita enviar mensajes a ese grupo de conversación. El activity en cuestión se llama “GUIGrupo”, e implementa una interfaz (definida en GUIGrupo.xml) que tiene el siguiente aspecto (Figura 124):



Figura 124: GUIGrupo.java. Interfaz gráfica

Se han marcados algunos elementos para detallarlos a continuación:

1. Título del grupo
2. Mensaje de otro miembro del grupo que no es el usuario, en este caso el usuario etiquetado como “mama”
3. Mensaje enviado por el usuario de la aplicación al grupo de conversación

Cuando enviamos un mensaje a una conversación de grupo, se crea un mensaje igual que si fuera una conversación pero con dos diferencias:

1. El destino del mensaje será el ID del grupo

2. La consulta HTTP con el mensaje se realizará a la página *recibirMensajeGrupo.jsp*

El proceso de recepción del mensaje en el servidor, y el envío a los usuarios correspondientes es:

1. Se reconstruye el mensaje recibido a través de la página *recibirMensajeGrupo.jsp* (Figura 125).
2. Se recupera el destino del mensaje, que será el ID del grupo al que pertenece
3. Se recuperan de la tabla usuario-grupo de la base de datos los regId de todos los usuarios que pertenecen a ese grupo (Figura 126).
4. Se envía un mensaje PUSH a través del servidor GCM a esos usuarios con el mensaje recibido. Para ello marcamos el tipo de mensaje como “tipo 3” para diferenciarlo de un mensaje normal a la hora de recibirlo en el cliente (Figura 127).

```
<body>
<%
System.out.println("Mensaje recibido. Contenido:");
String texto=null;
try{
    texto=request.getParameter("mensaje");
    System.out.println(request.getRemoteHost());
    GestorMensaje gm=new GestorMensaje(texto);
    gm.enviarMensajeGrupo();
    gm.imprimirPorPantalla();

    Long suma = gm.getSuma();
%>
<%=suma%>
<%

```

Figura 125: Recuperación de mensaje de grupo en el servidor. *recibirMensajeGrupo.jsp*

```

public void enviarMensajeGrupo(){
    try {
        Vector<Vector> v=Agente.getAgente().select("SELECT usuario FROM usuarios_grupo where grupo='"+m.getDestino());
        Grupo g=new Grupo();
        System.out.println("Grupo:"+v.get(0).get(3).toString());
        g.recuperarGrupo(Integer.parseInt(m.getDestino()));
        System.out.println("Hay que enviarlo a "+v.size());

        String origen=m.getOrigen();
        String titulo=g.getTitulo();
        GCMMensajeGrupo enviar=null;
        UsuarioBBDD ubd=new UsuarioBBDD();
        String idDestino="";

        for(int i=0;i<v.size();i++){
            if(ubd.existeNumEnBaseDeDatos(v.get(i).get(0).toString())){
                if(v.get(i).get(0).toString().equals(m.getOrigen())){

                    }else{
                        idDestino=ubd.getID(v.get(i).get(0).toString());
                        enviar=new GCMMensajeGrupo(origen, g,idDestino,m.getContenido());
                        try {
                            enviar.doPost();
                            System.out.println("enviado correctamente ");
                        } catch (ServletException e) {
                            // TODO Auto-generated catch block
                            System.out.println("no enviado!");
                        } catch (IOException e) {
                            System.out.println("error!");
                            e.printStackTrace();
                        }
                }
            }
        }

    }
}

```

Figura 126: recuperación de participantes en un grupo.

```

public void doPost() throws ServletException, IOException {
    System.out.println("Procedemos a enviar...");

    System.out.println("enviamos tipo 3");
    try {
    } catch (Exception e) {

        e.printStackTrace();
        return;
    }
    JSONArray mJSONArray = new JSONArray(Arrays.asList(this.participantes));
    Sender sender = new Sender(SENDER_ID);

    Message message = new Message.Builder()

        .collapseKey("")
        .timeToLive(30)
        .delayWhileIdle(true)
        .addData("tipo", "3")//tipo 3= mensaje grupo
        .addData("titulo", this.titulo)
        .addData("origen", origen)
        .addData("contenido", contenido)
        .addData("idgrupo", Integer.toString(idgrupo))

        .addData("participantes", mJSONArray.toString())
        .build();
}

```

Figura 127: Envío de mensaje de grupo del servidor al cliente. GCMMensajeGrupo.java

Por último, preparamos al cliente para recibir este tipo de mensajes GCM. Para ello se realizan las siguientes operaciones:

1. Se filtran los mensajes de tipo 3 (Figura 128)
2. Se recupera el contenido completo (Figura 129)
3. Se almacena en la base de datos y se actualiza la interfaz principal (Figura 130)

```
private void gestionarMensaje(Bundle extras) {  
    int tipo=Integer.parseInt(extras.getString("tipo"));  
  
    switch (tipo){  
        case 1:  
            leerMensajeNormal(extras);  
            break;  
        case 2://confirmacion grupo creado  
            leerMensajeConfirmacionGrupo(extras);  
            break;  
        case 3: //mensaje de grupo  
            System.out.println("recibido mensaje grupo");  
            leerMensajeGrupo(extras);  
            break;  
    }  
}
```

Figura 128: filtro de mensajes entrantes del servidor GCM

```

        titulo=extras.getString("titulo");
        origen=extras.getString("origen");
        idgrupo=extras.getString("idgrupo");
        contenido=extras.getString("contenido");
        participantes=extras.getString("participantes");
        ArrayList<String> listdata = new ArrayList<String>();
        JSONArray jArray = null;
        try {
            jArray = new JSONArray(participantes);
        } catch (JSONException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        if (jArray != null) {
            for (int i=0;i<jArray.length();i++){
                try {
                    listdata.add(jArray.get(i).toString());
                } catch (JSONException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }

        GestorMensaje gm =new GestorMensaje();
        gm.guardarMensajeGrupo(origen,idgrupo,contenido,titulo,listdata);
    }
}

```

Figura 129: Recuperación de mensaje de grupo en el cliente. Función leerMensajeGrupo. GCMIntentService.java

```

public void almacenarMensajeGrupo(String origen, String grupo, String contenido,
        GUIPanelPrincipal.actualizar(actualizarGrupo(origen,grupo,contenido,titulo,
        GrupoBBDD gbd=new GrupoBBDD(GUIPanelPrincipal.contexto));
        gbd.open();
        Boolean existeGrupo=false;
        existeGrupo=gbd.existe(grupo);
        if(existeGrupo){

        }else{
            Grupo g=new Grupo();
            g.setID(Integer.parseInt(grupo));
            g.setTitle(titulo);
            g.setParticipantes(participantes);

            gbd.insertarGrupo(g);
        }
        gbd.insertarMensaje(new Mensaje(origen,grupo,contenido));
        gbd.close();
    }
}

```

Figura 130: Almacenado de mensaje de grupo y actualización de la GUI

Planificación

Requisitos del cliente:

Requisito	Pendiente	En Proceso	Completo
RF C1			X
RF C2			X
RF C3			X
RF C4			X
RF C5			X
RF C6			X
RF C7			X
RF C8			X

Requisitos del servidor:

Requisito	Pendiente	En Proceso	Completo
RF S1			X
RF S2			X
RF S3			X
RF S4			X

Tras la implementación de este último ciclo, se han completado todos los requisitos funcionales. Mientras se han ido completando los R.F. también se han ido implementando los Requisitos no Funcionales (Sección 3.2.2.).

Aunque tras cada ciclo se ha probado el correcto funcionamiento del proyecto, el siguiente paso será diseñar un banco de pruebas para probar de forma objetiva todos los requisitos del proyecto, y su completo funcionamiento.

A continuación se muestra el diagrama UML de clases del sistema final. En la figura 131 se muestra el diagrama UML de clases del cliente de la aplicación. Como se aprecia está formado por 4 capas: Interfaz de usuario, dominio de la aplicación, persistencia de datos, y comunicación de red. En la figura 132 se muestra el diagrama de clases UML del servidor, formado también por 4 capas: interfaz de usuario, que son las páginas JSP, dominio, persistencia de datos, y red.

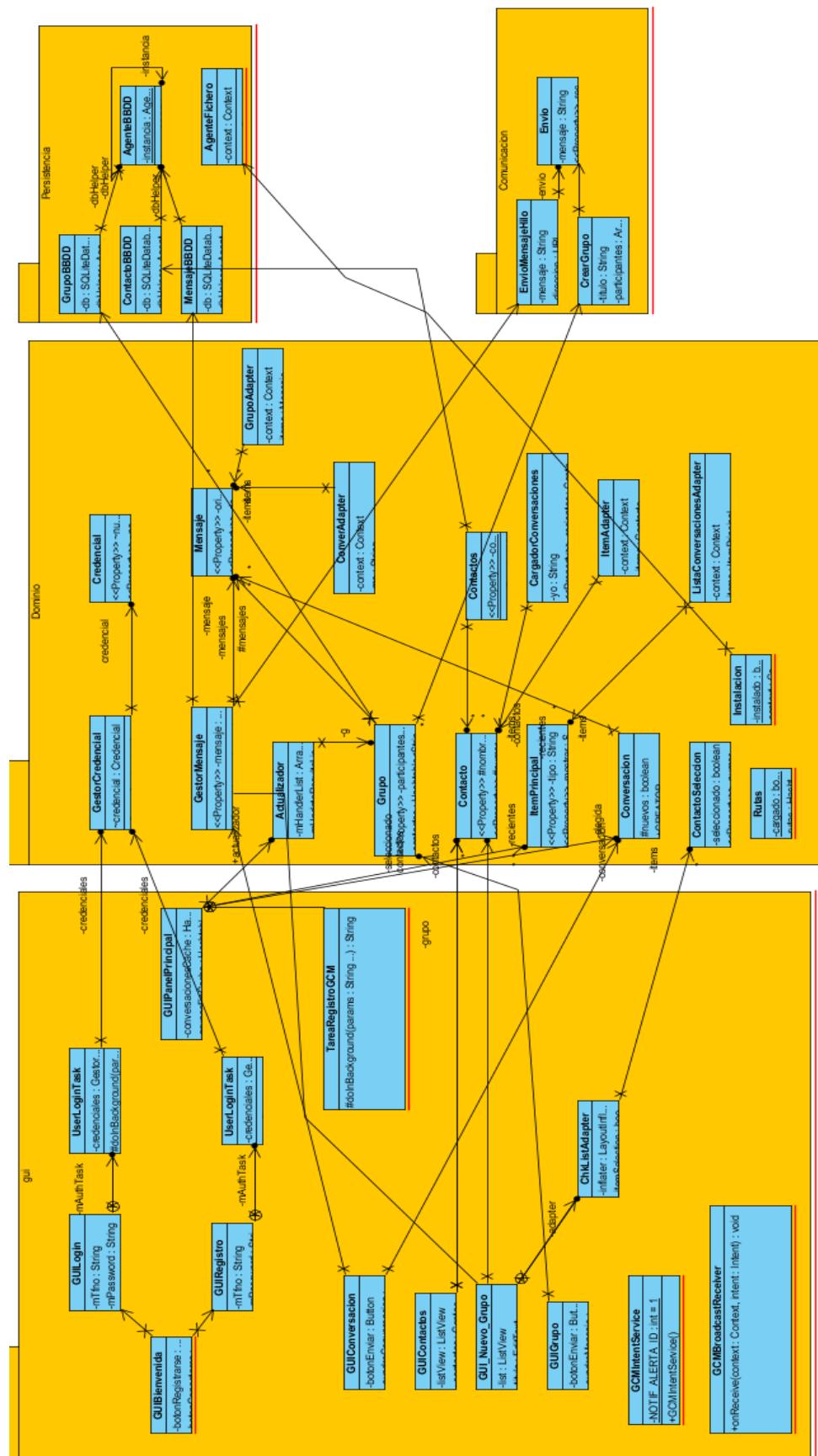


Figura 131: Diagrama de clases UML del Cliente. Ciclo 10

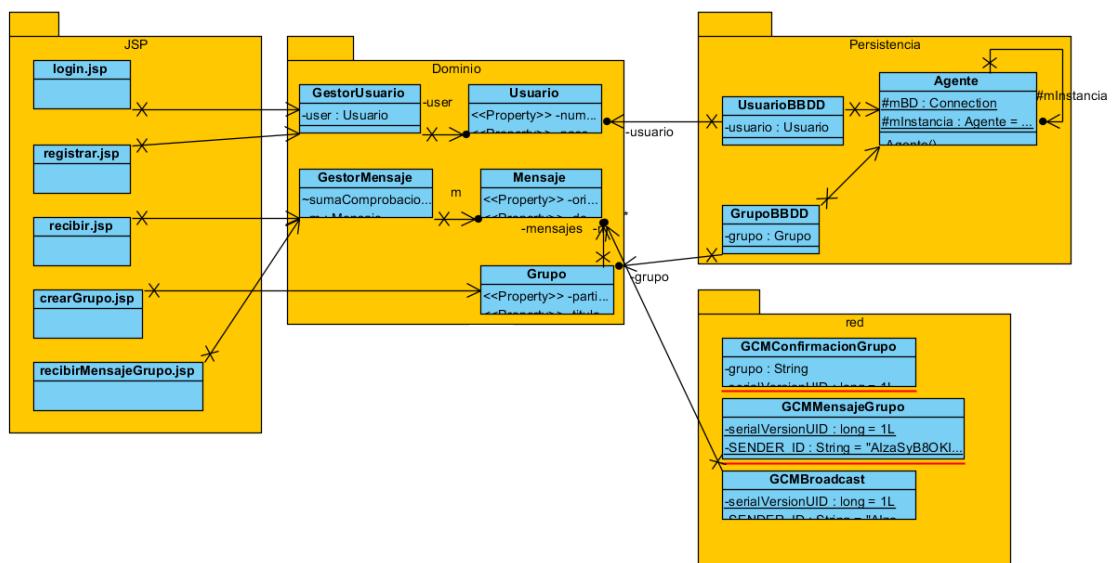


Figura 132: Diagrama de clases UML del servidor. Ciclo 10

5. Banco de pruebas

A continuación, una vez finalizados todos los ciclos de desarrollo, se ha diseñado un banco de pruebas para probar ciertas funcionalidades del sistema. Para ello se ha elegido utilizar casos de prueba de JUnit [42].

JUnit es un framework que sirve para generar un entorno de ejecución de clases Java controlado, con el fin de probar de forma aislada cada una de las funciones de las clases y poder asegurar su funcionamiento.

En el proyecto actual, el punto más crítico y por lo cual propenso a fallar son las conexiones entre el cliente y el servidor. Por ello, el presente banco de pruebas hace especial hincapié en las pruebas unitarias de esta parte del proyecto. Por ello, el proyecto de pruebas tendrá 3 conjuntos de pruebas, cada uno en una clase distinta:

- TestLogin: Es un conjunto de pruebas unitarias orientado a la gestión de credenciales y envío de estas al servidor
- TestRegistro: Este conjunto de pruebas pone casos extremos de intentos de registro con el fin de depurar posibles fallos.
- TestEnvioMensajes: En este conjunto de pruebas se comprueba que los mensajes enviados al servidor lleguen a este correctamente.

Las pruebas unitarias dentro de cada conjunto de pruebas se muestran a continuación (Figuras 133, 134 y 135):

```

public void testLoginCorrecto() {
    GestorCredencial gc=new GestorCredencial("616996366", "laguna", "");
    boolean res=gc.enviar();
    assertEquals(true, res);
}

public void testLoginIncorrecto() {
    GestorCredencial gc=new GestorCredencial("616996366", "passIncorrecta");
    boolean res=gc.enviar();
    assertEquals(false, res);
}

public void testLoginCorrectoYAnadirID() {
    GestorCredencial gc=new GestorCredencial("616996366", "laguna", "");
    gc.enviar();
    gc=new GestorCredencial("616996366", "laguna", "asdfwef434r34r4");
    boolean res=gc.enviar();
    assertEquals(true, res);
}

public void testIntentarRegistrarIDDeUserQueNoExiste(){
    GestorCredencial gc=new GestorCredencial("658856636", "laguna", "");
    gc.enviar();
    gc=new GestorCredencial("658856636", "laguna", "asdfwef434r34r4");
    boolean res=gc.enviar();
    assertEquals(false, res);
}

public void testLoginExtremo() {
    int contador=0;
    GestorCredencial gc=new GestorCredencial("616996366", "laguna", "");
    boolean res=false;
    for(int i=0;i<100;i++){
        res=false;
        res=gc.enviar();
        if(res){
            contador++;
        }
    }
    assertEquals(100, contador);
}

```

Figura 133: TestLogin

```

public void testRegistroCorrecto(){
    GestorCredencial gc =new GestorCredencial("6522332545", "holapruueba1", "");
    boolean res= gc.registrar();
    assertEquals(true, res);
}

public void testRegistroCorrectoConPasswordRara(){
    GestorCredencial gc =new GestorCredencial("6599853565", "*-/*-/f34f*-/*fg34g34", "");
    boolean res= gc.registrar();
    assertEquals(true, res);
}

public void testRegistroIncorrectoPorNumeroExistente(){
    GestorCredencial gc =new GestorCredencial("7899287745", "1234", "");
    gc.registrar();
    gc =new GestorCredencial("7899287745", "1234", "");
    boolean res= gc.registrar();
    assertEquals(false, res);
}

public void testRegistroYLogin(){
    GestorCredencial gc =new GestorCredencial("652252252", "1234", "");
    boolean res=gc.registrar();
    if(res){
        res= gc.enviar();
        assertEquals(true, res);
    }else{
        assertEquals(true, false);
    }
}

```

Figura 134: TestRegistro

```

public void testEnvioMensajeNormal() {
    final GestorMensaje gm=new GestorMensaje("hola","616996366","616200538");

    boolean res=gm.enviar(GUIPanelPrincipal.contexto,1,1);
    assertEquals(true, res);
}

public void testEnvioMensajeNormalAContactoInexistente() {
    final GestorMensaje gm=new GestorMensaje("hola","616996366","51618497984");

    boolean res=gm.enviar(GUIPanelPrincipal.contexto,1,1);
    assertEquals(true, res);
}

public void testEnvioMensajeNormalAContactoVacio() {
    final GestorMensaje gm=new GestorMensaje("hola","616996366","");
    boolean res=gm.enviar(GUIPanelPrincipal.contexto,1,1);
    assertEquals(true, res);
}

```

Figura 135: TestEnvioMensajes

Los resultados de los test no han sido completamente satisfactorios, si no nos han presentado pequeñas carencias del sistema que han sido subsanadas. Tras la mejora del sistema, los resultados han sido satisfactorios al 100% (Figura 136).

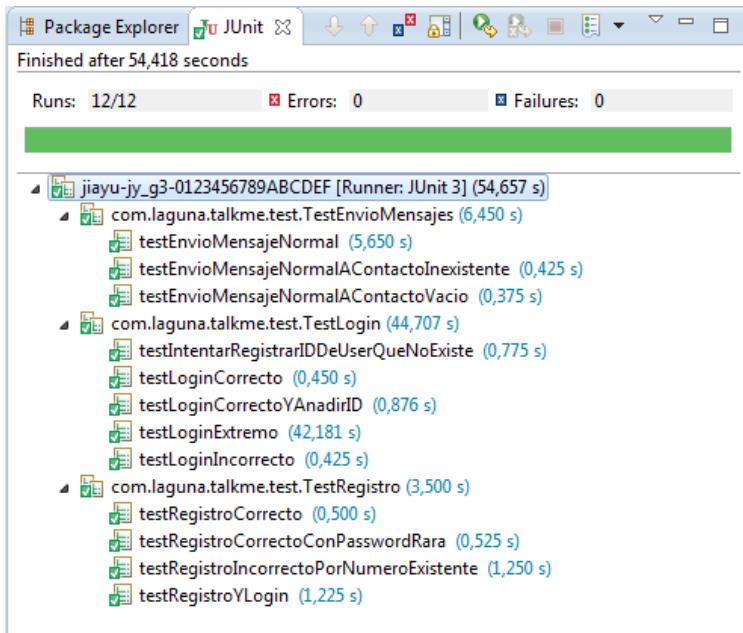


Figura 136: Resultado Banco de pruebas

6. Conclusiones

En esta sección se detallarán las conclusiones a las que se ha llegado tras la realización del presente TFG.

6.1. Objetivos funcionales

A continuación se muestran los objetivos planteados en la sección [3.2.] y se detalla la solución desarrollada en el presente TFG.

Requisito	Descripción	Solución
[RF C1]	Se debe mostrar al usuario una pantalla de acceso / registro la primera vez que se inicia el programa. Una vez que se haya accedido al sistema correctamente, el programa almacenará el usuario y la contraseña en el teléfono de manera que el acceso en posteriores ocasiones sea de forma automática.	Acceso, creación y modificación de ficheros de texto en Android utilizando la clase <i>InputStreamReader</i> y <i>OutputStreamWriter</i> . Clase <i>AgenteFichero.java</i>
[RF C2]	El usuario podrá ver una lista de sus contactos, y seleccionar aquel con el que quiere comenzar una conversación.	Acceso y gestión de la agenda interna del teléfono con la clase <i>ContentResolver</i> . Clase <i>Contactos.java</i>
[RF C3]	El usuario puede acceder a una lista de sus conversaciones.	Implementación de un Listview de Conversaciones en el Activity GUIPanelPrincipal utilizando un adaptador personalizado llamado <i>ListaConversacionesAdapter.java</i>
[RF C4]	El sistema almacenará los mensajes de forma persistente	Acceso y modificación de bases de datos SQLite en Android utilizando la superclase <i>SQLiteOpenHelper</i> en AgenteBBDD.java
[RF C5]	El usuario podrá cerrar de forma completa la aplicación, y no recibir mensajes a través de esta	Modificación de las funciones <i>onDestroy()</i> y <i>onBackPressed()</i> en GUIPanelPrincipal

		hasta que vuelva a ser abierta.
[RF C6]	El usuario podrá iniciar una conversación de grupo, en la que podrá elegir qué usuarios de su agenda participarán en ella.	Implementación en la clase Grupo, GUI Grupo y GUI_Nuevo_grupo
[RF C7]	El usuario podrá enviar un mensaje a un contacto	Envío de mensajes HTTP con parámetros utilizando las librerías de Apache HTTP. <i>Envio.java</i>
[RF C8]	El usuario podrá recibir mensajes de otros usuarios	Registro en servidor de notificaciones PUSH. Recepción de mensajes GCM utilizando un BroadcastReceiver. <i>GCMBroadcastReceiver.java</i>

Requisito	Descripción	Solución
[RF S1]	El Servidor recibirá los mensajes que todos los usuarios envíen a otros usuarios, y los almacenará en su base de datos	Recepción de peticiones HTTP POST con parámetros. Recuperación de información con JSON.
[RF S2]	El Servidor enviará mensajes a los usuarios que previamente ha recibido de otros usuarios.	Envío de mensajes HTTP al servidor GCM de Google con el mensaje y destinatario como parámetros
[RF S3]	El Servidor debe eliminar de su base de datos cada mensaje que ha sido recibido por el receptor.	No se llegan a almacenar los mensajes en la base de datos por lo que se cumple este requisito de forma natural.
[RF S4]	El servidor debe ser capaz de registrar nuevos usuarios en el sistema, a petición de los usuarios.	Gestión de peticiones HTTP POST con credenciales en los parámetros. Almacén de credenciales en base de datos mySQL

6.2. Objetivos didácticos

Para el desarrollo del presente TFG se han utilizado ciertas tecnologías de las cuales el alumno carecía de conocimientos y experiencia. Dicho desarrollo ha nutrido al alumno de conceptos y experiencia en tecnologías sin las cuales el resultado no habría sido satisfactorio. A continuación se detallan las tecnologías y lenguajes que el alumno ha utilizado y de las cuales ha requerido un entrenamiento previo para satisfacer las necesidades del proyecto.

- Arquitectura Android: Pese a utilizar Java como lenguaje de programación en la aplicación cliente, el sistema operativo Android tiene unas características que son necesarias conocer a la hora de llevar a cabo un desarrollo para esta plataforma móvil. Se desea destaca las siguientes características que han sido clave a la hora de desarrollar una aplicación completa para Android:
 - Ciclo de vida de un Activity
 - Uso de la memoria interna de Android
 - Gestión de hilos de ejecución
- Servidor JSP: Hasta la realización del presente TFG, el alumno no había tenido oportunidad de desarrollar un servidor completo a esta escala, así que el desarrollo del servidor utilizando tecnología JSP ha sido muy enriquecedora y se piensa que puede serle útil en un futuro cercano.
- Notificaciones PUSH: El sistema de comunicación entre el servidor y el cliente, siendo el servidor el que inicie la comunicación ha resultado ser una solución muy eficaz al problema de la recepción de mensajes. El alumno considera esta tecnología muy interesante y ha quedado muy sorprendido con el buen resultado.
- Gestión de Bases de Datos: No es un concepto nuevo, pero el alumno ha ampliado bastante sus conocimientos en gestión de bases de datos al haber utilizado SQLite en la aplicación cliente y MySQL en el servidor de la aplicación.

6.3. Trabajos futuros

Pese al correcto funcionamiento de la aplicación, la Ingeniería Informática está en constante desarrollo, y lo que hoy es novedoso mañana puede ser un sistema antiguo y desfasado. Por ello el actual TFG puede ser mejorado en muchos aspectos, y el alumno ofrece algunas alternativas para mejorarlo en un futuro. Son mejoras que no tenían cabida en el presente TFG y que pueden mejorar sustancialmente el resultado de la aplicación. Se ofrecen a continuación un conjunto de mejoras del proyecto, que no tienen que ser, ni mucho menos, las únicas:

- Cifrado de datos: El actual proyecto realiza la comunicación entre el cliente y el servidor de forma poco segura, enviando los datos como texto plano. Se piensa que cifrando estas comunicaciones el sistema será mucho más seguro.
- Almacén de credenciales: Actualmente el servidor almacena el par usuario-contraseña en una tabla de su base de datos. Se piensa que sería mucho mas seguro modificar este sistema y que cada usuario, en lugar de ser un registro en la base de datos del servidor, fuera un usuario de esta.
- Sistema de recuperación de contraseña: Si el usuario se olvida de su contraseña no es un problema hasta que intente instalar la aplicación en otro terminal. En este caso se proponen dos formas de recuperar la contraseña: La primera es modificar el formulario de registro, e incluir un campo email. En este caso, el servidor enviaría la contraseña al email que el usuario indicó al registrarse. La segunda opción es mediante el envío de un mensaje de texto SMS al número con el que el usuario se registró en el sistema. Este último sistema es quizás más costoso económicamente pero requiere menos modificaciones en el sistema de registro.
- Conversaciones guiadas: Actualmente el sistema permite dos tipos de conversaciones: uno a uno, y muchos a muchos. Se plantea una posible mejora el permitir al usuario realizar otros tipos de conversaciones, como por ejemplo una encuesta, en que los interlocutores no envían mensajes, sino que votan por una opción u otra, planteadas por el creador de la encuesta. Una vez finalizada la encuesta, se procedería a enviar el resultado a los interlocutores de este tipo de conversación.
- Transferencia de archivos: El envío y recepción de archivos es una función muy importante en casi cualquier sistema de mensajería. Se propone como posible mejora una ampliación del sistema actual incorporando un sistema de envío y recepción de archivos de cualquier tipo. A priori el sistema es lo suficientemente escalable para que esta ampliación sea sencilla. Se

recomienda utilizar MultipartEntity disponible en la librería de Apache HTTP [31]

7. Referencias

- [1] http://www.comscore.com/esl/Insights/Presentations_and_Whitepapers/2013/2013_Spanish_Digital_Future_in.Focus
- [2] http://www.idc.com/getdoc.jsp?containerId=prUS24676414
- [3] <http://tcanalysis.com/blog/posts/iv-estudio-sobre-mobile-marketing-de-iab-spain-y-the-cocktail-analysis>
- [4] <http://techcrunch.com/2014/03/12/hole-in-whatsapp-for-android-lets-hackers-steal-your-conversations/>
- [5] <http://www.abc.es/20121022/tecnologia/abci-aplicaciones-android-robo-datos-201210221554.html>
- [6] <http://www.securitybydefault.com/2011/03/whatsapp-y-su-seguridad-pwn3d.html>
- [7] http://www.securitybydefault.com/2011/06/lo-que-no-te-cuenta-whatsapp.html
- [9] <http://www.securitybydefault.com/2012/03/casi-10-millones-de-moviles-espanoles.html>
- [10] <http://www.securitybydefault.com/2012/09/whatsapp-spam-inundacion-y-robo-de.html>
- [11] <http://www.abc.es/medios-redes/20121129/abci-whatsapp-error-estado-contactos-201211290515.html>
- [12] http://cincodias.com/cincodias/2014/05/28/smartphones/1401278030_413773.html
- [13] <http://www.20minutos.es/noticia/2149043/0/whatsapp/caida-servicio/mundial/#xtor=AD-15&xts=467263>
- [14] <http://www.xatakamovil.com/aplicaciones/la-caida-del-servicio-de-whatsapp-provoca-tambien-la-caida-de-telegram-que-captaba-100-usuarios-segundo>

[15] James Gosling, Bill Joy, Guy Steele, y Gilad Bracha y Alex Buckley. 2013. The Java Lenguaje Specification (Java SE 7 Edition). Oracle. Disponible en <http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf> [Consulta: 27-11-2013]

[16] Android Developer. Documentación oficial de referencia de Android. Disponible en <http://developer.android.com/reference/packages.html> [Consulta: 27-11-2013]

[17] Crockford, D, 2006. The application/json Media Type for JavaScript Object Notation (JSON). RFC4627

[18] Institute of Electrical and Electronics Engineers, 1998. IEEE 830 - IEEE Recommended Practice for Software Requirements Specifications, ISBN 0-7381-0332-2 disponible en <http://standards.ieee.org/findstds/standard/830-1998.html> [Consulta: 27-11-2013]

[19] Rumbaugh, J; Jacobson, I; Booch, G. 2005. Unified Modeling Language User Guide, The 2nd Edition. Addison-Wesley, USA. ISBN: 978-0-321-26797-9

[20] <http://blog.whatsapp.com/613/500.000.000>

[21] <http://www.xatakamovil.com/aplicaciones/line-alcanza-los-300-millones-de-usuarios-registrados-al-acecho-de-whatsapp>

[22] <http://www.kantarworldpanel.com/es/Noticias/Android-ya-est-en-9-de-cada-10-nuevos-smartphones>

[23]
<https://developer.apple.com/library/mac/documentation/cocoa/conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html>

[24] <https://www.gnu.org/philosophy/free-sw.es.html>

[25] <https://www.kernel.org/>

[26] <http://www.freetype.org/>

[27] <https://code.google.com/p/dalvik/>

[28] [1] Institute of Electrical and Electronics Engineers, 1998. IEEE 830 - IEEE Recommended Practice for Software Requirements Specifications, ISBN 0-7381-0332-2 disponible en <http://standards.ieee.org/findstds/standard/830-1998.html> [Consulta: 27-11-2013]

[29] Boehm B. (1986). A Spiral Model of Software Development and Enhancement. ACM

[30] GSON - <https://code.google.com/p/google-gson/>

[31] APACHE HTTP <https://httpd.apache.org/>

[32] <http://www.mysql.com/>

[33] http://www.phpmyadmin.net/home_page/index.php

[34] <http://developer.android.com/reference/android/os/AsyncTask.html>

[35] <http://developer.android.com/reference/android/os/Parcel.html>

[36] http://cppcms.com/wikipp/en/page/benchmarks_all

[37] <https://source.android.com/devices/tech/dalvik/index.html>

[38] Schach, Stephen R. (2008). *Análisis y diseño orientado a objetos con el UML y el proceso unificado* Irwin/McGraw-Hill

[39] <http://albertolacalle.com/hci/interfaz.htm>

[40]
<https://developer.android.com/reference/android/support/v7/app/ActionBarActivity.html>

[41] <http://developer.android.com/google/gcm/index.html>

[42] <http://junit.org/>

8. ANEXOS

8.1. ANEXO 1: Manual de usuario

Registro de nuevo usuario

Requisitos previos:

- Instalación de la aplicación

Para registrarse como nuevo usuario en la aplicación se deben seguir los siguientes pasos:

1. En la pantalla de bienvenida se pulsa sobre el botón de registro indicado en la Figura 137:



Figura 137: Pantalla de bienvenida de la aplicación

2. Se rellena el formulario de la Figura 138 con el número de teléfono y una contraseña que se elija. No será necesario introducir la contraseña cada vez que se desee acceder a la aplicación, únicamente servirá para instalar la aplicación en un futuro en un terminal distinto.

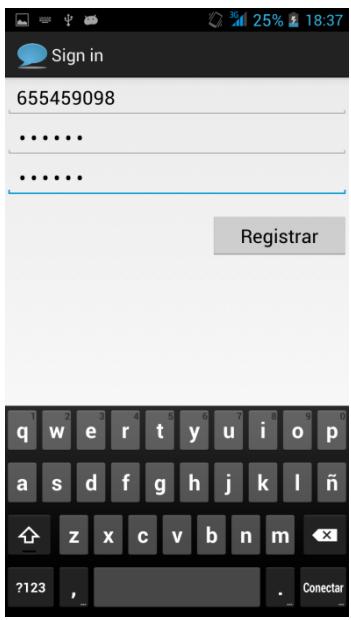


Figura 138: Interfaz de registro de la aplicación

3. Se pulsa el botón “Registrar” y tras unos segundos, si todos los datos son correctos se accederá a la interfaz principal del programa (Figura 139).

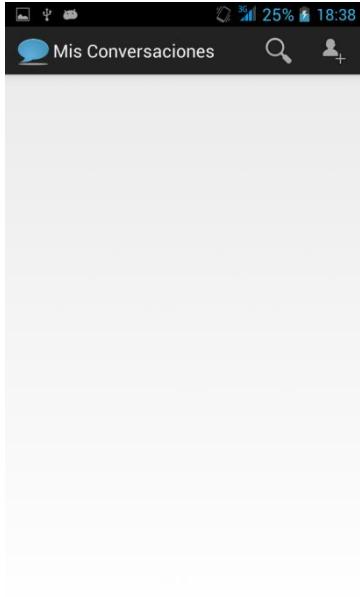


Figura 139: Interfaz principal de TalkMe

A partir de este momento, cada vez que iniciemos la aplicación (siempre y cuando no la desinstalemos previamente) apareceremos en la interfaz principal del programa sin necesidad de loguearnos manualmente.

Loguearse en la aplicación

Requisitos previos:

- Instalación de la aplicación
- Ser un usuario registrado en la aplicación

Este método de login en la aplicación solo es necesario en el caso de que la aplicación esté recién instalada y ya tengamos un usuario registrado en la aplicación, ya que si eres un nuevo usuario y te registras, el sistema te loguea automáticamente. No es una situación típica, el mejor ejemplo para esta situación sería un cambio de terminal del usuario de la aplicación, o una reinstalación del programa en el mismo terminal en el que se registró el usuario.

Los pasos a seguir para loguearse de forma manual en el sistema son:

1. En la pantalla de bienvenida se pulsa sobre el botón de “conectarse” indicado en la Figura 140:



Figura 140: Pantalla de bienvenida. Opción Conectarse

2. Se rellena el formulario de la Figura 141 con el número de teléfono y la contraseña que se eligió cuando se registró el usuario. Al igual que cuando se registra un usuario, no será necesario introducir la contraseña cada vez que se

desea acceder a la aplicación; esta será almacenada y se realizará un login automático.

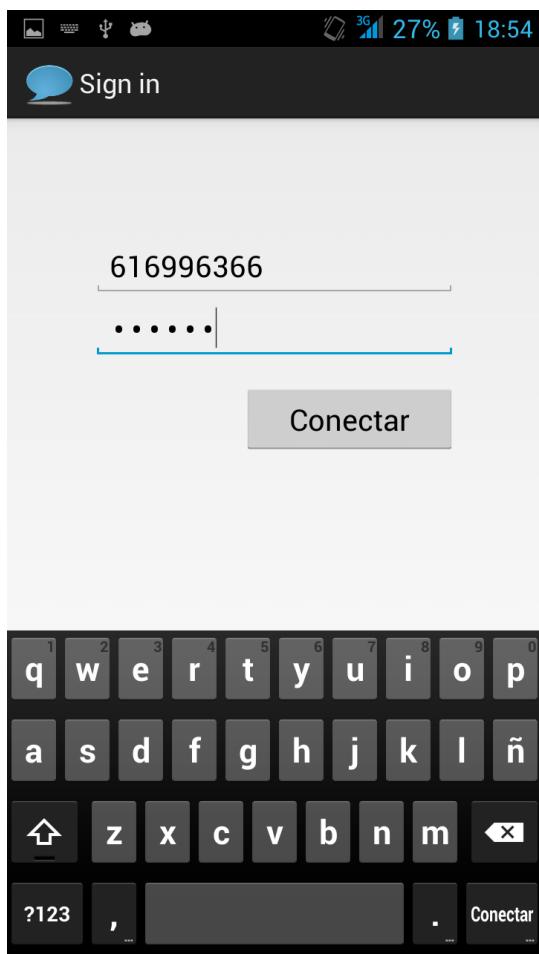


Figura 141: Formulario de login

3. Pulsamos el botón “conectar” de la Figura 141 y tras unos segundo, si los datos son correctos, se nos mostrará la interfaz principal de la aplicación (Figura 139)

Iniciar una conversación

Requisitos previos:

- Instalación de la aplicación
- Ser un usuario registrado en la aplicación
- Estar logueado en la aplicación, y por tanto tener acceso a la interfaz principal del programa (Figura 139)

Para iniciar una conversación con un contacto, los pasos son los siguientes:

1. En la interfaz principal de la aplicación (Figura 139) pulsamos el botón de nueva conversación (Figura 142)

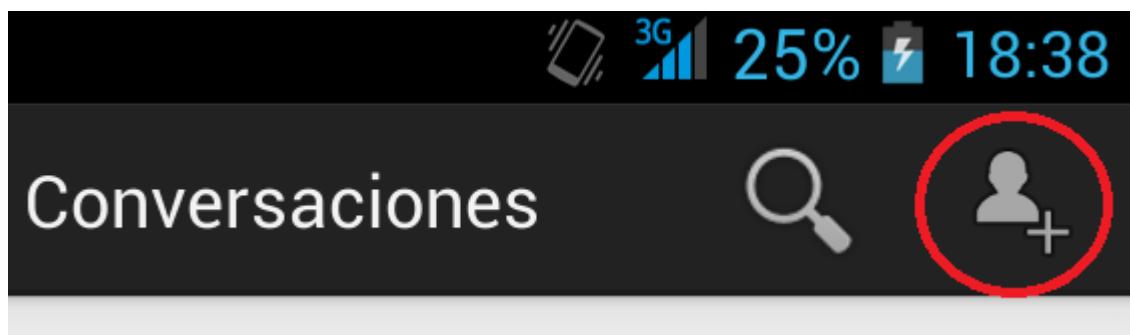


Figura 142: Botón de nueva conversación

2. A continuación se nos despliega una lista de los contactos disponibles de nuestra agenda (Figura 143). Pulsamos sobre uno de ellos para iniciar una conversación con él.



Figura 143: Selección de contacto para iniciar conversación

3. Se creará una conversación vacía en la interfaz principal a la que accederemos pulsando sobre ella (Figura 144)

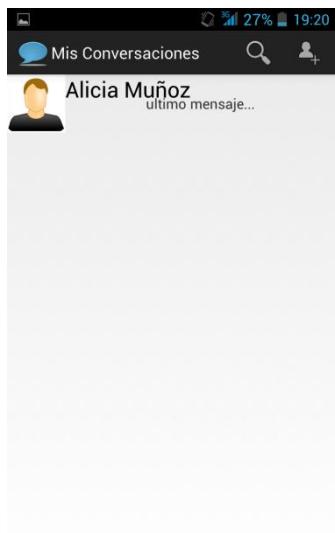


Figura 144: Panel principal con conversación vacía

4. Se nos abrirá una interfaz de conversación con el contacto elegido (Figura 145), y podremos enviarle un mensaje escribiéndolo en la parte inferior y pulsando el botón enviar.

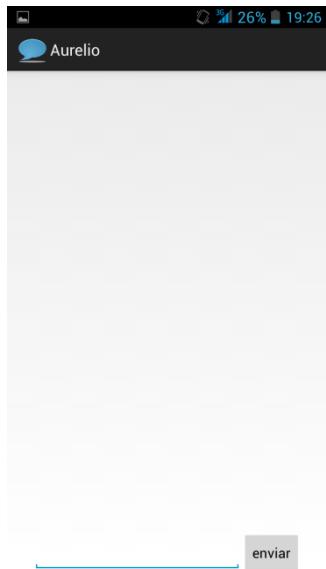


Figura 145: Conversación vacía

5. El mensaje aparecerá en la parte superior de la interfaz con un tic verde (Figura 145), que nos indicará que el mensaje ha sido enviado correctamente.



Figura 146: Mensaje enviado correctamente

6. La conversación permanecerá en la interfaz principal y para seguir enviando mensajes al mismo contacto habrá que repetir los pasos 3-6

Crear un grupo de conversación

Requisitos previos:

- Instalación de la aplicación
- Ser un usuario registrado en la aplicación
- Estar logueado en la aplicación, y por tanto tener acceso a la interfaz principal del programa (Figura 139)

A continuación se describen los pasos a seguir para crear un grupo de conversación. Un grupo de conversación es una conversación entre 3 o más usuarios, los cuales envían y reciben mensajes a todos los usuarios del grupo a la vez.

1. En el menú principal, seleccionamos la opción de “Nuevo Grupo” (Figura 147)



Figura 147: Botón Nuevo Grupo

2. En la pantalla que aparece (Figura 148) seleccionamos los usuarios que se quieren añadir al grupo de conversación, y en la parte superior se inserta el nombre que se desea asignar al grupo.

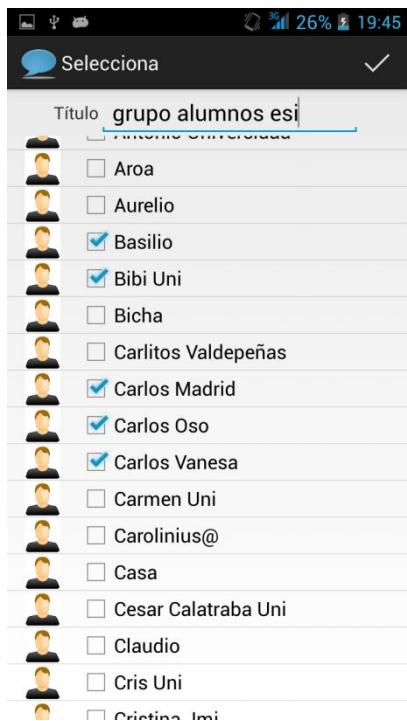


Figura 148: Creación de nuevo grupo

3. Pulsamos en el botón de la esquina superior derecha para confirmar los datos.
4. Tras unos segundos se nos devolverá a la interfaz principal donde tendremos el grupo creado, pero aún vacío de mensajes (Figura 149). De momento, y hasta que el creador del grupo no escriba un primer mensaje, los demás participantes no tendrán constancia de la creación del grupo.



Figura 149: Grupo creado

5. Para enviar mensajes al grupo pulsamos sobre el nombre del grupo en el panel principal
6. Se nos abrirá una ventana de conversación en grupo (Figura 150)

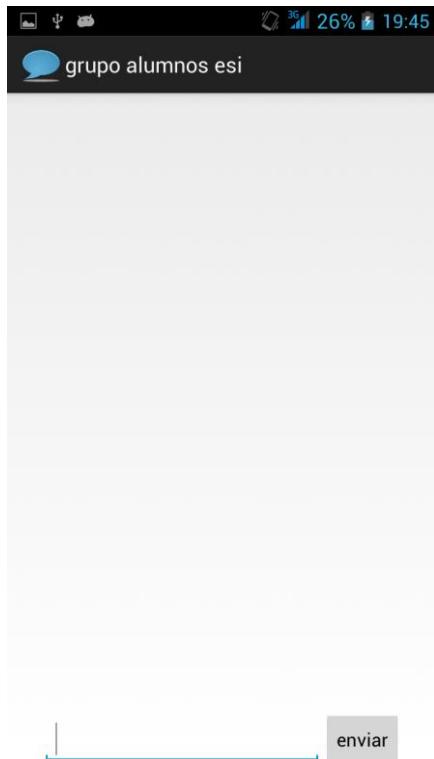


Figura 150: Interfaz de conversación en grupo vacía

7. Escribimos un mensaje en la parte inferior de la pantalla y pulsamos en enviar.
8. El mensaje aparecerá en la parte superior de la pantalla como enviado (Figura 151), y los demás usuarios del grupo lo habrán recibido y podrán participar en la conversación.

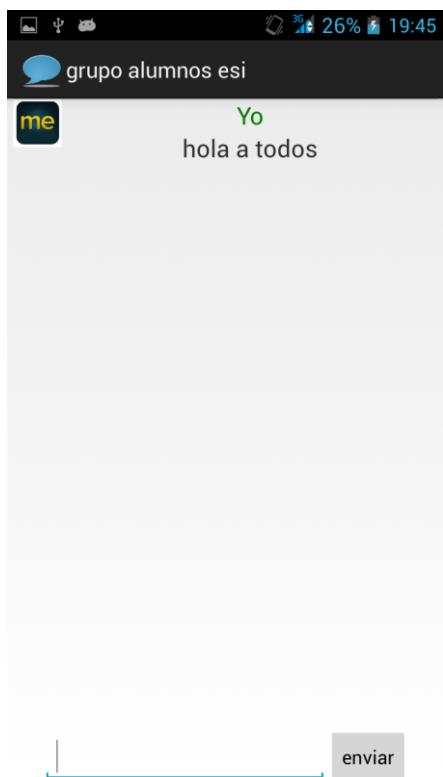


Figura 151: Interfaz de conversación en grupo con el primer mensaje del creador

8.2. ANEXO 2: Instalación del servidor

El servidor actualmente se encuentra instalado en un ordenador del alumno, pero si se desea cambiar su ubicación los pasos a seguir serían los siguientes:

1. Instalar la base de datos suministrada por el alumno en el propio servidor.
2. Indicar en el código del servidor Agente.java (Figura 152) la nueva ruta de la base de datos

```
1
20     private static void conectar() throws Exception {
21         //bbdd local
22         String url="jdbc:mysql://localhost:3306/servidorbbdd";
23         //bbdd internet
24         //String url="jdbc:mysql://db4free.net:3306/bbddchat";
25         String driver="com.mysql.jdbc.Driver";
26         Class.forName(driver);
27         mBD=DriverManager.getConnection(url, "root", "");
28     }
29 }
```

Figura 152: Modificación de la ruta de la base de datos del servidor

3. Generar un nuevo archivo *servidor.war*, ya que el suministrado por el alumno contiene la ruta de la base de datos que se utiliza en su servidor.
4. Montar el nuevo archivo WAR en el servidor web donde se desee instalar el servidor de la aplicación. El alumno lo monta en *C:\xampp\tomcat\webapps*
5. Modificar el fichero de rutas *\$/res/raw* (primera línea) con la IP de la nueva localización del servidor (Figura 153).

```
1 server http://81.44.73.161:8080 listo
2 recibir /Servidor1/recibir.jsp listo
3 instalacion instalacion.cnf listo
4 log /Servidor1/login.jsp listo
5 registro /Servidor1/registro.jsp listo
6 get /Servidor1/getMensajes.jsp listo
7 existe /Servidor1/comprobar.jsp listo
8 crearGrupo /Servidor1/crearGrupo.jsp listo
9 confirmarRecepcion /Servidor1/confirmarRecepcion.jsp listo
10 pedirPendientes /Servidor1/pedirPendientes.jsp listo
11 recibirGrupo /Servidor1/recibirMensajeGrupo.jsp listo
12 confirmarGrupo /Servidor1/confirmarRecepcionGrupo.jsp listo
13 |
```

Figura 153: fichero rutas. Modificación de IP

6. Se compila el proyecto del cliente y se vuelve a instalar en los terminales Android, ya que el archivo *.apk suministrado tiene asociada la IP de la antigua situación del servidor.

