UNIVERSIDAD CENTRAL DE VENEZUELA

FACULTAD DE CIENCIAS

ESCUELA DE COMPUTACIÓN

TRABAJO ESPECIAL DE GRADO

Título

Desarrollo de una librería basada en tecnología push para el envío de mensajes a dispositivos móviles Android.

Realizado por

Ronald Morey **C.I.:** 18.712.570

Tutor: Prof. Robinson Rivas

Mayo, 2015

Índice

Índice de Figuras

Índice de Tablas

Introducción

En la actualidad el mercado de aplicaciones móviles ha crecido enormemente. Cada vez más personas tienen teléfonos celulares y dispositivos de alta tecnología a precios accesibles. Esto hace que a su vez los fabricantes de aplicaciones para dispositivos móviles se vean obligados a satisfacer el mercado desarrollando programas que mantengan actualizados a los usuarios en tiempo real. Este tipo de aplicaciones se basan en programas o servicios que están en constante comunicación con servidores y estos se encargan de informar al usuario de manera inmediata el arribo de algún mensaje, notificación, estatus o cualquier otro tipo de información. Todo esto sin que el usuario tenga que hacer la solicitud de dicha información. Este tipo de comunicación es conocido como Push.

Ejemplos del uso de este tipo de comunicación se puede ver en aplicaciones básicas del día a día como correo electrónico, mensajerías instantáneas, redes sociales entre otras. Aplicaciones en donde el arribo de la información en tiempo real suele ser los que las hace marcar la diferencia.

A pesar de que en el mercado se ofrecen varias alternativas para que los desarrolladores no tengan que reinventar la rueda al momento de querer implementar este tipo de comunicación, las limitaciones suelen ser incómodas para algunos que quieren soluciones más independientes.

El objetivo fundamental de este Trabajo Especial de Grado consiste en el desarrollo de unas librerías basadas en tecnología push para el envío de mensajes a dispositivos Android sin necesidad de agentes externos.

Este Trabajo Especial de Grado está estructurado en seis (6) capítulos:

En el primer capítulo se lleva a cabo la propuesta de Trabajo Especial de Grado, la cual contempla el problema presentado, la solución empleada para desarrollar las librerías basadas en push, la justificación de dicho problema, los objetivos y el alcance de esta propuesta.

El segundo capítulo se refiere a toda la parte conceptual con algunos conceptos en los que se basa la tecnología push, se habla un poco sobre el Sistema Operatio Android y de las alternativas existentes para crear aplicaciones que se puedan comunicar vía push.

En el tercer capítulo se plantea, diseña y desarrolla la solución propuesta con las especificaciones necesarias para su uso.

En el cuarto capítulo se diseñan y desarrollan tres aplicaciones que harán uso de la librería desarrollada en el capitulo tres para demostrar el alcance y funcionamiento de las mismas.

El capítulo 5 consiste en la elaboración de pruebas a las librerías utilizando las aplicaciones desarrolladas en el capítulo cuatro.

Finalmente en el capítulo cinco se presentan las conclusiones del presente trabajo así como también las referencias.

Capítulo 1: Planteamiento del Problema

En este capítulo se presenta la situación actual, justificación e importancia de crear desarrollar unas librerías independientes que permitan mantener la comunicación push entre dispositivos clientes Android y un servido, el objetivo general del Trabajo Especial de Grado, sus objetivos específicos, el alcance de la aplicación y los usuarios potenciales.

1.1 Situación Actual

Actualmente existen varias alternativas para el uso de esta tecnología en cada plataforma móvil. Algunas de estas alternativas son pagas, otras son gratuitas. Una de estas alternativas para Android es utilizar el servicio GCM de Google.

El servicio Google Cloud Messaging for Android (GCM) permite el envío de datos a dispositivos Android vía Push a través de sus servidores, este servicio se encarga de de encolar los mensajes y el envío de los mismos a los dispositivos Android que se hayan registrado.

GCM ofrece los API necesarios para la conexión, registro y envío de mensajes desde los servidores de terceros hacia los dispositivos finales.

A pesar de que el servicio GCM de de Google es eficiente rápido y fácil de usar requiere una conexión de datos a sus servidores. Por lo tanto toda la información enviada pasará a través de ellos.

Adicionalmente se pueden encontrar otros servicios que proporcionan el uso de librerías[1] para el manejo de mensajes push, pero estás librerías suelen ser pagas o utilizan internamente el servicio GCM, por lo que hace falta tener una cuenta de Google que es necesaria para crear un proyecto en la consola de desarrolladores de google (Developers Console).

[1] Como por ejemplo pushwoosh, (https://www.pushwoosh.com/) y pushover (https://pushover.net/) que se detallan más adelante

Una de las limitaciones del uso del servicio GCM de Google es que es necesaria una conexión a Internet para el envío y la recepción de los mensajes, el uso de los servidores de Google para en el envío y recepción de mensajes también puede ser considerado una desventaja para aquellos que son muy precavidos en cuanto al tema de privacidad y confidencialidad de datos. Una de las limitaciones más importantes para el uso del GCM es que el dispositivo Android que haga uso del servicio debe tener como mínimo la versión 2.2 del sistema operativo. Para los dispositivos que tengan una versión entre la 2.2 y la 4.0 se debe tener configurada y el usuario debe haber iniciado sesión con sus credenciales de Google en el dispositivo cliente. Y como limitante obligatoria cada dispositivo debe tener instalado el Google Play Service para su funcionamiento. Por lo que versiones personalizadas o ROMS modificadas de Android que no incluyan este servicio no podrán utilizar el GCM.

Debido a esto en este Trabajo Especial de Grado se ha planteado desarrollar librerías que permitirán el uso de mensajes push desde servidores a Dispositivos Android sin la necesidad de una cuenta de Google, o servidores de terceros.

1.2 Justificación e importancia

A pesar de que existen soluciones nativas, gratuitas, y pagas para mantener comunicación push con las aplicaciones la mayoría de ellas cuenta con una gran limitante que es la dependencia de agentes externo para utilizar este servicio. Debido a que son servicios ofrecidos al público y pueden ser utilizados por millones de personas se imponen limitaciones que en casos muy específicos pueden limitar la funcionalidad de una aplicación. Lo planteado anteriormente refleja la importancia de contar con un servicio completamente independiente.

1.3 Objetivo General

Diseño desarrollo y evaluación de dos librerías que permitan mantener una conexión entre un equipo cliente y un equipo servidor que se encuentre en cualquier tipo de Red (Red Privada/ Red Pública) usando tecnología push.

1.4 Objetivos específicos

- Diseñar y desarrollar una librería que pueda ser utilizada por cualquier aplicación/servicio de Windows, que se encargue de recibir y mantener las conexiones con los dispositivos que se encuentren conectados a la aplicación/servicio que la utilizan esta librería. Esta librería se debe encargar de encolar y recibir los mensajes enviados hacia y desde los dispositivos.
- Proveer métodos que permitan la manipulación de los datos que serán enviados y recibidos por los dispositivos.
- Proveer métodos que permitan el envío de datos a dispositivos individuales y a todos los dispositivos.
- Diseñar e implementar un esquema de Base de datos que se encargue de almacenar los mensajes encolados y los mensajes enviados hacia los dispositivos, permitiendo la opción de consultar si el mensaje fue entregado o no.
- Diseñar y desarrollar una librería que pueda ser utilizada por cualquier aplicación Android, que se encargue de realizar y mantener la conexión hacia el servidor, recibir y enviar mensajes al mismo.
- Diseñar y desarrollar 3 casos de estudio en donde se utilicen las librerías desarrolladas.

1.5 Alcance

El alcance de este trabajo especial de grado es el diseño y desarrollo de las librerías que permitan realizar la comunicación push entre varios dispositivos clientes y un servidor. Y la realización de una prueba de concepto con tres aplicaciones que utilicen la comunicación push con el uso de estas librerías.

1.6 Potenciales usuarios

Los usuarios a los que va dirigido la solución propuesta son desarrolladores de aplicaciones Android en java que estén interesados en agregar la funcionalidad de comunicación push a sus aplicaciones sin necesidad de dependencia de agentes/empresas externas.

Capítulo 2: Marco Teórico

2.1 Tecnología Push

La tecnología Push se refiere a un estilo de comunicación en la red en donde el servidor es el encargado de notificar y hacer llegar al cliente datos asociados a alguna información recibida al instante. Debido a que es el servidor el encargado de entregar la información al cliente esta llega de forma casi “instantánea” y en tiempo real, siempre y cuando el dispositivo cliente se encuentre conectado al servidor.

A un bajo nivel la tecnología consiste en realizar y mantener una conexión abierta con el servidor Push y cuando información nueva llega esta es enviada directamente a través de la conexión existente, ahorrando en primera instancia el proceso de abrir un socket y realizar la conexión con el cliente. Es cómo dejar la puerta siempre abierta para que un envío llegue directamente a la casa en lugar de tener que tocar la puerta, esperar a que alguien abra, reciba el paquete, y luego vuelva a cerrar la puerta.

2.2 Tecnología Pull

Al contrario de Push la tecnología Pull consiste en consultar/consumir desde el servidor la información que se desea obtener. Para dar la impresión de noticias o información en tiempo real el cliente debe realizar consultas (requests) frecuentes al servidor preguntando si existe información nueva. Esto es lo que desde hace tiempo se hace con las páginas web. Un usuario consulta y descarga desde el servidor la página que desea para ver si esta contiene nueva información.

2.3 TCP

El "protocolo de control de transmisión" ('Transmission Control Protocol', TCP) está pensado para ser utilizado como un protocolo 'host' a 'host' muy fiable entre miembros de redes de comunicación de computadoras por intercambio de paquetes y en un sistema interconectado de tales redes.

TCP es un protocolo orientado a la conexión, fiable y entre dos extremos, diseñado para encajar en una jerarquía en capas de protocolos que soportan aplicaciones sobre múltiples

redes. TCP proporciona mecanismos para la comunicación fiable entre pares de procesos en computadoras 'host' ancladas en redes de comunicación de computadoras distintas, pero interconectadas. Se hacen muy pocas suposiciones sobre la fiabilidad de los protocolos de comunicación por debajo de la capa de TCP. TCP sólo supone que puede acceder a un servicio de transmisión de datagramas simple, aunque en principio poco fiable, de los protocolos del nivel inferior. En principio, TCP debería ser capaz de operar encima de un amplio espectro de sistemas de comunicaciones que incluye desde conexiones por cables fijos ('hard-wired conections') hasta redes de intercambio de paquetes o redes de circuitos conmutados.

TCP encaja en una arquitectura de protocolos en capas justo por encima del protocolo de internet, protocolo básico que proporciona un medio para TCP de enviar y recibir segmentos de longitud variable de información envuelta en "sobres" de datagramas de internet. El datagrama de internet proporciona un medio de direccionar TCPs de origen y de destino situados en redes diferentes. El protocolo de internet también trata con la fragmentación y el reensamble de segmentos de TCP que sean necesarios para conseguir el transporte y la entrega sobre múltiples redes y las puertas de enlace que las interconectan. El protocolo de internet también lleva información sobre la prioridad, clasificación de seguridad y compartimentación de los segmentos de TCP, de tal forma que esta información pueda ser comunicada de extremo a extremo entre múltiples redes [1]. En la Figura 1 se puede ver el protocolo TCP en la estructura de capa de protocolos.

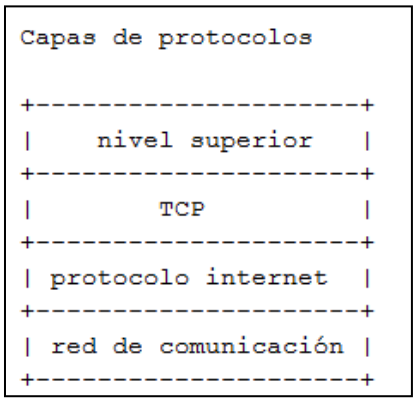


Figura 1 Protocolo TPC en la capa de protocolos

El lado cliente de una conexión realiza una apertura activa de un puerto enviando un paquete *SYN* inicial al servidor como parte de la negociación en tres pasos. En el lado del servidor (este receptor también puede ser una PC o alguna estación terminal) se comprueba si el puerto está abierto, es decir, si existe algún proceso escuchando en ese puerto, pues se debe verificar que el dispositivo de destino tenga este servicio activo y esté aceptando peticiones en el número de puerto que el cliente intenta usar para la sesión. En caso de no estarlo, se envía al cliente un paquete de respuesta con el bit *RST* activado, lo que significa el rechazo del intento de conexión. En caso de que sí se encuentre abierto el puerto, el lado servidor respondería a la petición *SYN* válida con un paquete *SYN*/*ACK*. Finalmente, el cliente debería responderle al servidor con un *ACK*, completando así la negociación en tres pasos (*SYN*, *SYN*/*ACK* y *ACK*) y la fase de establecimiento de conexión. Una vez que la conexión ha sido establecida se puede comenzar la transferencia de datos entre el cliente y el servidor como se indica en la Figura 2.

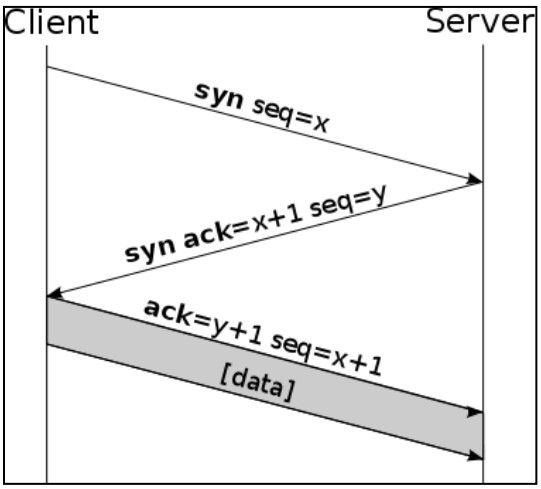


Figura 2 Establecimiento de la conexión [2]

2.4 Android

Android es un sistema operativo basado en el núcleo Linux diseñado originalmente para dispositivos móviles, tales como teléfonos inteligentes, pero que posteriormente se expandió su

desarrollo para soportar otro tipo de dispositivos como tablets, reproductores mp3, netbook, pc, televisores, etc.

Es desarrollado por OpenHanset Alliance dirigida por Google.

Android provee un SDK con herramientas y APIs requeridos para el desarrollo de aplicaciones utilizando varios lenguajes de programación, el más popular de ellos, Java.

Historia

Android, Inc. Fue fundado en Palo Alto, California (Estados Unidos de América) en Octubre de 2003, por Andy Rubin, Rich Miner, Nick Sears, y Chris White. Para ese entonces se sabía muy poco sobre las funciones de Android, solo se conocía que desarrollaban software para teléfonos móviles.

En Agosto del 2005 Google adquirió Android Inc. Haciendo de Android Inc. Pase a ser propiedad de Google. En Google el equipo dirigido por Andy Rubin desarrolló una plataforma móvil basada en el kernel Linux.

El 5 de Noviembre de 2007 se dio a conocer *Open Hanset Alliance (Alianza para los Dispositivos Móviles Abiertos)* un consorcio de varias compañías entre las que se encuentra, Broadcom Corporation, Google, HTC, Intel, LG, Marvell Technology Group, Motorola, Nvidia, Qualcomm, Samsung Electronics, Sprint Nextel, T-Mobile y Texas Instruments. La idea del *Open Hanset Alliance* es desarrollar estándares abiertos para dispositivos móviles. Ese mismo día se dio a conocer el primer producto Android, una plataforma móvil construida utilizando el Kernel de Linux Versión 2.6.

Antecedentes

Android ha tenido una gran cantidad de actualizaciones desde la primera versión del Sistema operativo, cada versión corrigiendo bugs y agregando nuevas características. Generalmente cada versión es desarrollada bajo un nombre código de un elemento relacionado con postres.

Todos los dispositivos con sistema operativo Android son dispositivos de tercera generación que incluyen características básicas como: Explorador Web, soporte para cámara, cliente de correo electrónico, GPS, SMS, MMS, Reproductor Multimedia, Wallparers, Bluetooth, WiFi, entre otras funciones básicas. Como el sistema operativo Android pertenece a Google este ha incorporado aplicaciones para el uso de sus servicios, estas aplicaciones son: Gmail, Google Contacts, Google Calendar, Google Maps, Google Sync, Google Search, Google Talk y un reproductor de video para Yoube, además de poseer el Android Market en donde los usuarios podrán descargar aplicaciones para los dispositivos.

Actualmente en el mercado Android se encuentra distribuido mayormente con equipos que poseen una versión de del Sistema operativo 4.x.x. Estas versiones incluyen nuevas características y mejoras. En la Figura 3 se puede observar la distribución de equipos (Activos) Android en el mercado.

Version	Codename	API	Distribution
2.2	Froyo	8	0.4%
2.3.3 - 2.3.7	Gingerbread	10	6.4%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	5.7%
4.1.x	Jelly Bean	16	16.5%
4.2.x		17	18.6%
4.3		18	5.6%
4.4	KitKat	19	41.4%
5.0	Lollipop	21	5.0%
5.1		22	0.4%

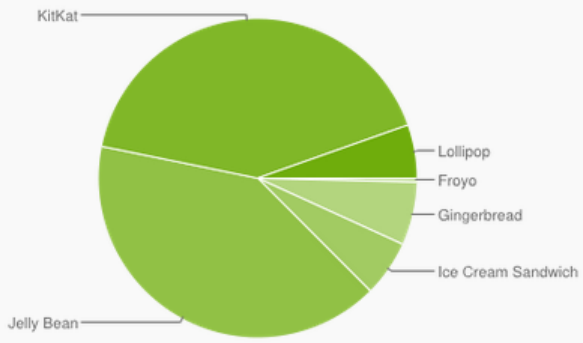


Figura 3 Distribución de equipos Activos Android por versión [3]

Android 2.2 Froyo

2.2.2 basado en el Kernel de Linux 2.6.32 liberado el 20 de mayo de 2010.

En esta versión se agregó la funcionalidad de USB tethering y hostspot Wi-Fi, actualización del Android Market, soporte para instalar aplicaciones en la memoria externa, la inclusión del motor V8 javascript de chrome en el explorador y mejoras extras en el sistema.

Android 2.3.X Gingerbread

Basado en el kernel de Linux 2.6.35 liberado oficialmente el 6 de Diciembre de 2010

Esta versión incluye una mejora significativa en el teclado virtual, funcionalidad copy/paste mejorada, un nuevo administrador de descargas, soporte SIP VoIP, soporte para voz y video para Google Talk, actualizaciones para el cliente de correo Gmail y mejoras al sistema.

Android 3.x Honeycomb

Basado en el kernel de Linux 2.6.36 liberado oficialmente el 22 de febrero de 2011 para la versión 3.0, 10 de Mayo del 2011 para la versión 3.1, 15 de julio de 2011

Esta plataforma cuenta con un nuevo diseño en la interfaz grafica, lo que la hace ideal solamente para dispositivos tablets, mejora en el soporte multitasking, la aceleración por hardware y soporte para procesadores multi núcleos, conectividad para accesorios USB, lista de aplicaciones recientes, soporte para teclado externo y dispositivos apuntadores (mouse), soporte para joysticks y gamepads, y compatibilidad para aplicaciones que no han sido diseñadas para tablets.

Android 4.x - Ice Cream Sandwich

Liberado oficialmente el 19 de Octubre de 2011.

Esta versión nueva y mejorada incluye una nueva interfaz de usuario, separación de widgets en nuevas pestañas, facilidad para crear carpetas con estilos drag and drop, accesos directos personalizables, función de screenshot integrada, búsqueda offline, diccionario del teclado mejorado, habilidad para acceder a las aplicaciones directamente desde la pantalla bloqueada, copy/paste mejorado, desbloqueo de la pantalla por reconocimiento facial, nuevas pestañas para el explorador, soporte para hasta 16 pestañas, Sincronización automática del explorador con los favoritos de chrome, capacidad para cerrar programas que están utilizando datos en el background, Editor de fotos, nuevo diseño en la galería, widgets ajustables, Wi-Fi Direct y mejoras en el sistema.

Características

Algunas de las características de la plataforma Android son:

Framework Android: Incluye un entorno de desarrollo con emuladores de dispositivos, herramientas para depuración y análisis de rendimiento de software.

Maquina virtual Dalvik: Maquina virtual optimizada para dispositivos móviles. Esta Maquina virtual será definida más adelante.

Explorador Web integrado: Explorador web basado en el motor de renderización WebKit[2] y el motor de Javascrip V8 que utiliza google chrome.

Gráficos Optimizados: Gráficos que funcionan con una librería personalizada 2D y gráficos 3D basados en OpenGL 1.0 (Aceleración de hardware opcional).

Soporte Multimedia: Soporta formatos de audio, video e imágenes comunes (MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF).

Telefonía GSM: (depende del hardware)

Bluetooth, EDGE, 3G y Wi-Fi: (Depende del hardware)

Cámara, GPS, Brújula, y Acelerómetro: (depende del hardware).

Soporte para streaming: Streaming RTP/RTSP (3GPP PSS, ISMA), descarga progresiva de HTML (HTML5 <video> tag). Adobe Flash Streaming (RTMP) es soportado mediante el Adobe Flash Player.

Multi-táctil: Android tiene soporte nativo para pantallas multi-táctiles.

Videollamada: Android soporta videollamada através de Google Talk desde su versión HoneyComb.

Características basadas en voz: La búsqueda en Google a través de voz está disponible como "Entrada de Búsqueda" desde la versión inicial del sistema.

[2] Webkit es un motor web open source que actualmente es utilizado por varios exploradores como chrome, safari .

Tethering: Android soporta tethering, que permite al teléfono ser usado como un punto de acceso alámbrico o inalámbrico (todos los teléfonos desde la versión 2.2, no oficial en teléfonos con versión 1.6 o superiores mediante aplicaciones disponibles en el Android Market, por ejemplo PdaNet).

Multitarea: Multitarea real de aplicaciones está disponible.

Arquitectura

El Sistema operativo Android puede ser subdividido en una arquitectura de 5 capas como se muestra en la Figura 4. El Kernel de Linux, las Librerías, el entorno de ejecución de Android, el Framework de Aplicaciones y las aplicaciones de usuario final. [4]

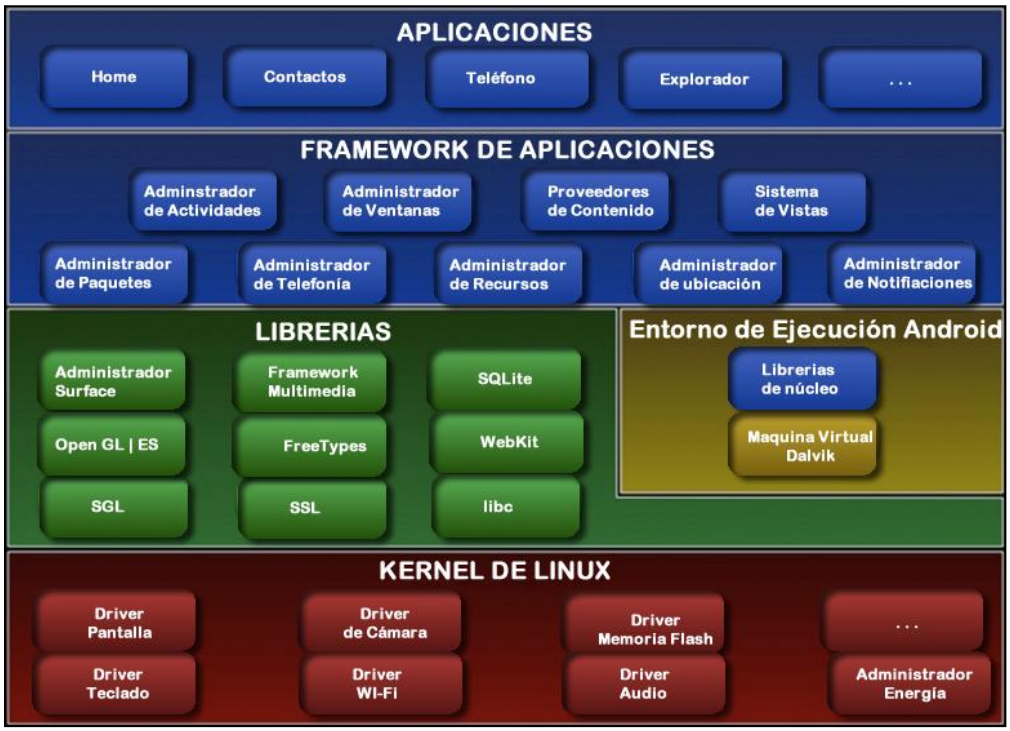


Figura 4 Arquitectura Android

Kernel de Linux:

Internamente Android utiliza Linux para la administración de la memoria, procesos, comunicaciones y otros servicios del sistema operativo, el kernel de Linux es transparente al usuario final y las aplicaciones no harán llamadas a Linux directamente.

Librerías nativas:

En esta capa el kernel contiene librerías nativas de Android las cuales son accedidas desde el kernel. Estas librerías compartidas están escritas en C o C++, compiladas para un uso particular en la arquitectura del hardware utilizado por el teléfono, y son preinstaladas por el fabricante.

Algunas de las librerías nativas más importantes son las siguientes:

- *Surface Manager:* Android utiliza un administrador de ventanas similar a Windows Vista o al administrador de ventanas compiz, pero es mucho más simple. En lugar de dibujar directamente en el buffer de la pantalla, los comandos para dibujar van directamente a imágenes (bitmaps) fuera de la pantalla que son combinadas con otras imágenes que el usuario ve. Esto permite al sistema crear toda clase de efectos interesantes tales como ver a través de las ventanas y elegantes transiciones.
- *Gráficos 2D y 3D:* elementos de 2 y 3 dimensiones pueden ser combinados en una sola interfaz de usuario con Android. Esta librería utilizará hardware 3D si el dispositivo lo tiene o un software de renderización rápida si no posee hardware 3D.
- *Codecs para contenido multimedia*: Android puede reproducir y grabar audio en una variedad de formatos incluyendo AAC,AVC(H.264), H.263,MP3, y MPEG-4.
- *Base de datos SQLite:* Android incluye esta base de datos ligera para almacenamiento interno.
- *Motor Web:* Para la rápida renderización de contenido HTML, Android utiliza la librería WebKit.

Entorno de ejecución Android:

Incluye la maquina virtual Dalvik y las librerías básicas de java.

Dalvik es una maquina virtual diseñada y escrita por Dan Bornstein en Google. Dalvik ha sido escrita de forma que un dispositivo puede correr múltiples máquinas virtuales de forma eficiente. Dalvik ejecuta archivos en el formato Dalvik Executable (.dex), el cual está optimizado

para memoria mínima. La Máquina Virtual está basada en registros y corre clases compiladas por el compilador de Java que han sido transformadas al formato.dex por la herramienta incluida "dx".

Framework de Aplicaciones:

Esta capa proporciona los bloques de alto nivel que se utilizaran para elaborar una aplicación. El framework viene preinstalado con Android, pero también se puede extender con componentes propios si se necesitan.

Las partes más importante del Framework de aplicaciones son:

- Administrador de Actividades: Esto controla el ciclo de vida de las aplicaciones y proporciona una “pila de actividades” para la navegación del usuario.
- Proveedores de contenido Estos objetos encapsulan datos que necesitan ser compartidos entre aplicaciones, como por ejemplo la aplicación de Contactos.
- Administrador de recursos: Los recursos son cualquier tipo de elementos o archivos que van con los programas que no son código (imágenes, xml,fuentes, etc.).
- Administrador de Notificaciones: Eventos como el arribo de un mensaje, apuntes, alertas y más pueden ser presentadas en la barra de notificaciones del usuario.

Aplicaciones:

Son todas las aplicaciones que ve y ejecuta el usuario final.

Las aplicaciones base incluyen un cliente de correo electrónico, programa de SMS, calendario, mapas, navegador, contactos y otros.

Fundamentos de las Aplicaciones

Las aplicaciones de Android que se desarrollan en Java Utilizan el SDK de Android.

El SDK de Android compila el código (con cualquier cantidad de datos y archivos) en un Paquete de Android, un archivo con extensión .apk. Todo el código en un archivo .apk es

considerado una aplicación. Y así este archivo puede ser instalado en cualquier dispositivo que utilice Android.

Una vez instalado en el dispositivo cada aplicación vive en su propia caja de seguridad con las siguientes características:

- El SO Android es un sistema multi usuario, por lo cual cada aplicación es un usuario diferente.
- Por defecto el sistema asigna un *User Id* único para cada aplicación (El ID solo es usado por el SO y es desconocido por la aplicación).
- Cada proceso tiene su propia Máquina Virtual, así que una aplicación se ejecuta aisladamente de las otras.
- Por defecto cada aplicación se ejecuta en su propio proceso de Linux. Android inicia el proceso cuando cualquiera de las aplicaciones necesita ser ejecutada; luego la aplicación se cierra si no es necesaria o si el sistema debe recuperar memoria para otras aplicaciones.

De este modo el sistema implementa el principio del mínimo privilegio. En el que cada aplicación por defecto solo tiene acceso a los componentes que requiere para hacer su trabajo.

Sin embargo hay formas en las que varias aplicaciones pueden compartir datos con otras aplicaciones y para que las aplicaciones puedan acceder a servicios del sistema.

Es posible configurar dos aplicaciones para que estas compartan el mismo User ID, en este caso estas aplicaciones serán capaces de acceder a los archivos de la otra.

Una aplicación puede pedir permiso para acceder a datos en el dispositivo como por ejemplo los SMS, la tarjeta extraíble (SD), cámara, bluetooth, y más.

Componentes de la aplicación

Las aplicaciones de Android están compuestas de uno o más componentes (Actividades, Servicios, proveedores de contenido, y receptores broadcast).

- **Actividades:** una actividad representa una sola pantalla con una interfaz de usuario.
- **Servicios:** Un servicio es un componente que se ejecuta en Segundo plano para realizar operaciones largas o para trabajar con procesos remotos.
- **Proveedores de contenido**: Un proveedor de contenido administra y comparte un conjunto de datos de la aplicación.
- **Receptores Broadcast:** Un receptor broadcast es un componente que responde a anuncios realizados por el sistema.

Funcionamiento de Android

En Sistemas Operativos estándar de escritorio como Linux o Windows, el usuario puede tener múltiples aplicaciones corriendo y visibles en diferentes ventanas al mismo tiempo, una de las ventanas tiene el focus (cursor) del teclado, pero sin embargo todos los programas son iguales. Un usuario puede fácilmente intercambiar entre ventanas, pero es responsabilidad del usuario mover las ventanas alrededor para ver lo que están haciendo las otras y cerrarlas cuando ya no se necesitan.

Sin embargo, en los Sistemas Operativos para telefonía, este esquema de trabajo consume demasiados recursos. Por ello, Android no trabaja de esa manera. En Android, hay una aplicación en primer plano, que típicamente toma toda la pantalla excepto por la barra de estado. Cuando el usuario enciende el teléfono, la primera aplicación que ve es la aplicación de la pantalla principal (Home application).

Cuando el usuario ejecuta una aplicación Android la inicia y la trae a primer plano. Desde esa aplicación el usuario puede invocar otras aplicaciones, u otras pantallas de la misma aplicación, y estas a la vez otras pantallas y aplicaciones. Todas esas aplicaciones y pantallas son guardadas en la *pila de actividades* por el Administrador de Actividades. En cualquier momento el usuario puede presionar el botón de retroceso para volver a la pantalla o aplicación previa en la pila. Desde el punto de vista del usuario funciona con el historial en un navegador web. Presionando el botón de volver, vuelve a la página anterior.

Actividades

Internamente, cada interfaz de usuario es representada por una clase *Activity*. Cada Actividad tiene su propio ciclo de vida. Una aplicación es una o más actividades más un proceso de Linux que los contiene. [5]

Una Actividad es un componente de la aplicación que proporciona una pantalla en la cual el usuario puede interactuar para hacer algo como marcar un número telefónico, tomar una foto, enviar un email, o ver un mapa. A cada actividad se le asigna una ventana en la cual se dibuja la interfaz de usuario. La pantalla normalmente llena la pantalla completa, pero puede ser más pequeña y posicionarse encima de otras ventanas.

Una aplicación usualmente consiste en múltiples actividades que están relacionadas entre si. Típicamente, una actividad en una aplicación es especificada como la actividad "principal", que es presentada al usuario cuando este ejecuta la aplicación por primera vez.

Cuando una actividad es detenida por el inicio de una nueva, se notifica de su cambio de estado a través del ciclo de vida de las actividades mediante los métodos *callback*. Hay muchas funciones *callback* que una actividad puede recibir, debido a su cambio de estado (si el sistema lo está creando, deteniéndolo, retomándolo, o destruyéndolo) y cada llamada a estas funciones provee la oportunidad de realizar acciones especificas dependiendo del cambio de estado. Por ejemplo, cuando una actividad es detenida, esta debería liberar objetos muy pesados, tales como conexiones a red o bases de datos. Cuando la actividad se reanuda se pueden volver a obtener los recursos necesarios y retomar la actividades que fueron interrumpidas. Estas transiciones de estados son parte del ciclo de vida de una Actividad y la transición de estos estados puede apreciarse en la Figura 5.

Durante su ciclo de vida, cada actividad de un programa de Android puede estar en varios estados. A la hora de desarrollar una aplicación no se tiene el control de estos estados, todos estos estados son administrados por el sistema. Sin embargo hay métodos que informan los cambio de estados, estos son los métodos con formato onXX(). Estos métodos se pueden sobrecargar para que realicen una actividad en el momento adecuado.

onCreate(Bundle): Este método es llamado la primera vez que una actividad inicia. Se puede utilizar para realizar operaciones de inicialización, tales como creación de la interfaz grafica. *onCreate()* tiene un solo parámetro que es null u otra información de estado previamente almacenado con el método *onSaveInstanceState().*

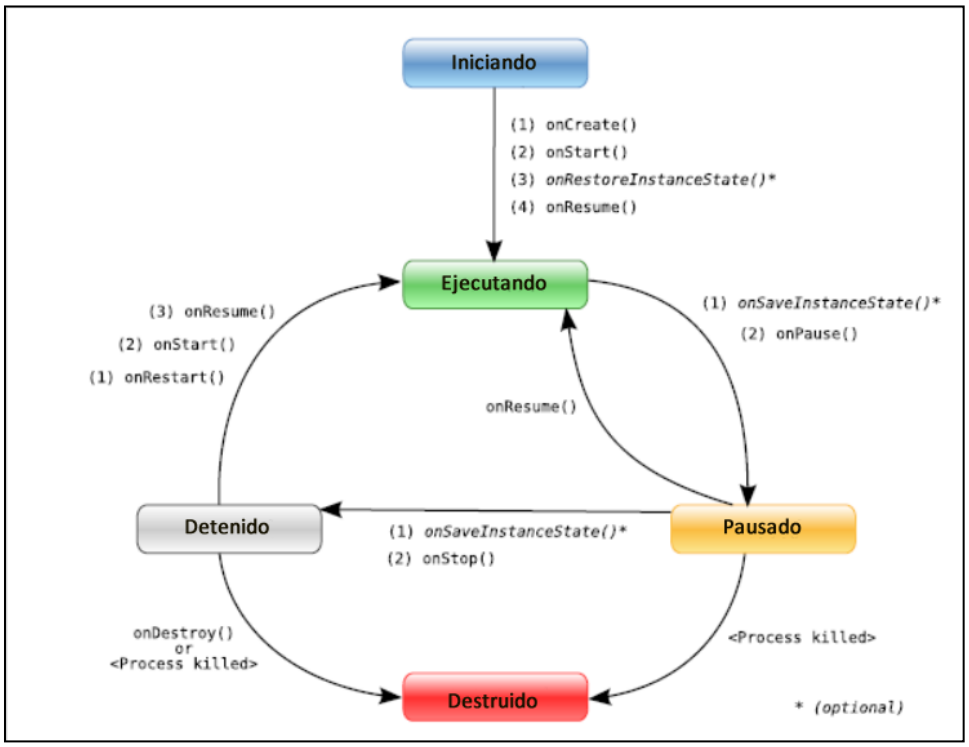


Figura 5 Ciclo de Vida de una Actividad

onStart(): Este método indica que la actividad está a punto de ser mostrada al usuario.

onResume(): Este método es llamado cuando la actividad puede comenzar a interactuar con el usuario. Este es un buen lugar para colocar animaciones y música.

onPause(): Este método se ejecuta cuando la actividad está a punto de ir a segundo plano, usualmente porque otra actividad ha sido lanzada delante de esta. En este método se debe guardar el estado persistente del programa, tales como base de datos y registros que están siendo editados.

onStop(): Este método es llamado cuando una actividad ya no es visible al usuario y no va a ser usada por un tiempo. Si la memoria es muy poca *onStop()* podría nunca ser llamado(el sistema simplemente termina el proceso).

onRestart(): si este método es llamado, indica que la actividad está siendo mostrada nuevamente al usuario después de haber estado detenida.

onDestroy(): ese método es llamado antes de que la actividad sea destruida. Si la memoria es muy poca, *onDestroy()* podría nunca ser llamado (el sistema termina el proceso).

onSaveInstanceState(Bundle): Android llamará este método para permitir que la actividad guarde un estado, tal como la posición del cursos en un campo de texto. Normalmente este método no necesita ser sobrecargado porque por defecto se almacenan todos los estados de la interfaz de estado automáticamente.

onRestoreInstanceState(Bundle): Este método es llamado cuando la actividad está siendo reinicializada de un estado previamente guardado por el método onSaveInstanceState(). La implementación por defecto restaura el estado de la interfaz de usuario.

Agregar Actividades a un proyecto

Cada actividad que se agregue al proyecto debe ser referenciada en el archivo de configuración *AndroidManifest.xml.* En la Figura 6 se muestra un ejemplo de cómo se deben referenciar las Actividades en el archivo Manifest.

```
<activity android:name=".Agregar" android:label="@string/agregar"></activity>
<activity android:name=".Listar" android:label="@string/listar"></activity>
<activity android:name=".Editar" android:label="@string/editar"></activity>
<activity android:name=".Ver" android:label="@string/ver_persona"></activity>
```

Figura 6 Actividades referenciadas en el archivo AndroidManifest

El atributo *android:name* hace referencia al nombre de la clase (archivo .java) y el atributo *android:label* será el titulo de la ventana que tendrá la actividad.

Tareas y pila de actividades

Una aplicación normalmente contiene múltiples actividades.

Una tarea es una colección de actividades que con las que el usuario interactúa para ejecutar cierto trabajo. Las actividades están ordenadas en una pila (llamada *pila de actividades)* en el orden en que cada actividad fue abierta.

La pantalla principal del dispositivo (*Home screen*) es el lugar en el que normalmente las aplicaciones suelen iniciarse. Cuando un usuario toca el icono de una aplicación para iniciarla las tareas de dicha aplicación vienen a pasar a primer plano.

Cuando la actividad actual inicia otra actividad la nueva actividad es colocada en el tope de la pila y es la que aparece en la pantalla principal. La actividad anterior permanece en la pila pero es detenida (*stopped)*. Cuando una actividad se detiene el sistema retiene el estado actual de la interfaz de usuario. Cuando el usuario presiona el botón de retroceso *(Back button),* la actividad actual es desapilada de la *pila de actividades* (la actividad es destruida) y la actividad previa es retomada (el estado previo de la Interfaz de Usuario es restaurado). Las actividades en la pila nunca son reordenadas, solo son apiladas y desapiladas. [6] En la Figua 7 se muestra el ciclo de vida de las actividades en la pila de actividades.

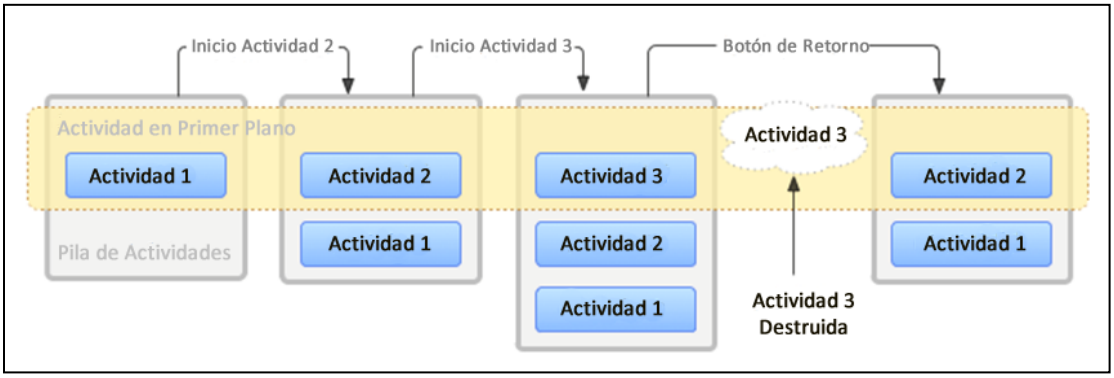


Figura 7 Pila de actividades

Una tarea es una unidad que puede ser movida a segundo plano. Cuando un usuario inicia una nueva tarea o va a la pantalla principal *(Home Screen)* a través del botón principal *(Home Button)* esta tarea es llevada a segundo plano. Mientras está en segundo plano, todas las actividades en la tarea son detenidas, pero su *pila de actividades* se mantiene intacta. Una tarea simplemente pierde la pantalla principal mientras otra tarea ocupa ese lugar. Una tarea puede volver a primer plano y así el usuario puede retomarla como la dejó. Como ejemplo

tenemos que supongamos que la tarea actual A (Tarea A) tiene tres actividades en su pila de actividades, dos de las cuales están actualmente en uso. El usuario presiona el botón principal *(Home Button),* entonces el sistema inicia una nueva tarea para esa aplicación (Tarea B) con su propia pila de actividades. Después de interactuar con la aplicación el usuario vuelve a presionar el botón principal *(Home Button)* y selecciona la aplicación que inició la tarea A (Tarea A), ahora la tarea A vuelve a primer plano, y todas las actividades en la pila de actividades están intactas y la actividad que se encuentra en el tope de la pila se reanuda. En este punto el usuario puede volver a la tarea B presionando el botón principal *(Home Button)* y seleccionando el icono de la aplicación que inició la tarea B. De este modo se realiza el Multitasking en Android. En la Figura 8 la Tarea B interactúa con el usuario en primer plano, la tarea A se mantiene en Segundo plano esperando a ser reanudada.

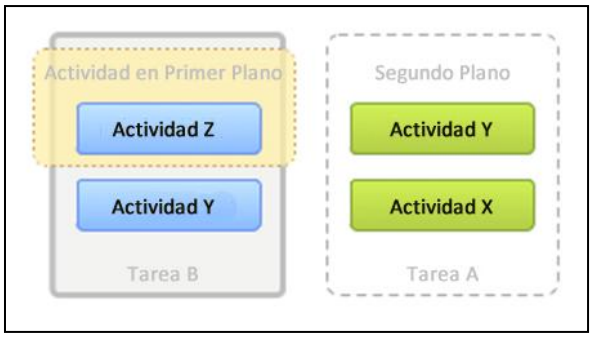


Figura 8 Tareas en segundo plano.

Múltiples tareas pueden estar en background al mismo tiempo, pero si el usuario tiene muchas tareas en segundo plano, el sistema puede comenzar a destruir actividades que estén en segundo plano para así recuperar memoria, causando que el estado de las actividades en background se pierda.

Como las actividades en la *pila de actividades* nunca son reordenadas, si una aplicación permite que el usuario pueda iniciar una cierta actividad desde más de una actividad esto causará que se cree una nueva instancia de la esa actividad (en lugar de traer la actividad iniciada anteriormente), como se muestra en la Figura 9.

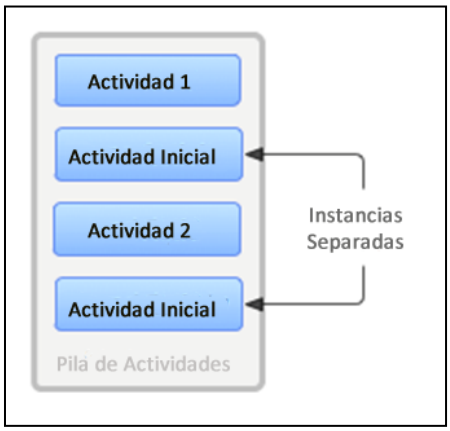


Figura 9 Una actividad es instanciada varias veces.

Servicios

Son componentes de aplicación que puede ejecutar operaciones de larga ejecución en segundo plano y no proporciona una interfaz de usuario**.** Otro componente de una aplicación puede iniciar un servicio y este continuara ejecutándose en segundo plano aunque el usuario cambie a otra aplicación. [7]

Adicionalmente, un componente puede unirse a un servicio para interactuar con él, e incluso realizar comunicación entre procesos. Por ejemplo, un servicio puede manejar operaciones de red, reproducir música, realizar operaciones de entrada/salida o interactuar con un generador de contenido, todo esto desde un segundo plano.

Un Servicio puede tomar dos formas:

- ***Iniciado(Started):***

  Un servicio esta iniciado cuando el componente de una aplicación(por ejemplo una actividad) comienza llamando el método startService(). Una vez iniciado, un servicio puede ejecutarse en segundo plano indefinidamente, aún incluso si el componente que lo inicio ha sido destruido (Finalizado/liberado de memoria). Usualmente un servicio iniciado realiza una sola operación y no retorna un resultado a la actividad que lo llamo.

- ***Vinculado(Bound):***

  Un Servicio está vinculado cuando el componente de una aplicación se vincula a el llamado al método bindService(). Un servicio vinculado ofrece una interfaz cliente-

servidor que permite a los componentes interactuar con el servicio, enviar peticiones, obtener resultados e incluso realizar comunicación entre procesos.

Un proceso vinculado solo se ejecutará mientras la aplicación que está vinculado a él se ejecute.

Intents

Tres de los cuatro tipos de componentes (Actividades, servicios y receptores broadcast) son activados por un mensaje asíncrono llamado *Intent*.

Los *Intents* enlazan los componentes individuales en tiempo de ejecución (se pueden ver como mensajeros que solicitan una petición de otros componentes), si componente pertenece a una aplicación o a otra.

Proveedores de contenido

Los proveedores de contenido almacenan y recuperan datos haciéndolos accesible a todas las aplicaciones. Estos proveedores son la única manera de compartir datos a través de las aplicaciones; no hay un área no común que todos los paquetes de Android puedan acceder.

Seguridad

En Android cada aplicación se ejecuta en su propio proceso de Linux. El hardware le prohíbe a un proceso acceder a la memoria de otro. Además, cada aplicación tiene asignado un ID de usuario especifico. Cualquier Archivo creado no puede ser leído o modificado por otras aplicaciones.

Aparte de esto, el acceso a ciertas operaciones criticas están restringidas, y se debe especificar para pedir permiso de acceso en el archivo llamado Android-Manifest.xml. Cuando una aplicación es instalada, el instalador autoriza o no basado en el certificado, y si es necesario le pregunta al usuario.

Algunos de los permisos más comunes que se necesitan son:

- **INTERNET**: Permiso de acceso a Internet.
- **READ_CONTACTS**: Lee los contactos del usuario.

- **WRITE_CONTACTS**: Escribe en los contactos del usuario.
- **RECEIVE_SMS**: Monitorea Mensajes de Texto.

La especificación de esta permisología se muestra en el Figura 10:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.google.android.app.myapp" >
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
</manifest>
```

Figura 10 Permisología en el archivo AndroidManifest

2.5 Push Android

Android ofrece varias alternativas para la implementación de mensajería push en sus dispositivos, la más popular y conocida es el servicio *Google Cloud Messaging for Android* (GCM) que permite el envío de datos a dispositivos Android vía push mediante el uso de sus servidores, este servicio se encarga de de encolar los mensajes los mensajes y el envío de los mismos a los dispositivos Android que se hayan registrado.

GCM ofrece los API necesarios para la conexión, registro y envío de mensajes de los servidores clientes[3] revisar los formatos de superíndice y referencias hacia los dispositivos finales.

GCM es un servicio gratuito que ayuda a los programadores a enviar datos desde servidores a Aplicaciones Android en dispositivos móviles que utilicen esta tecnología, y así mismo estos dispositivos pueden enviar Datos a los Servidores de Google. Esta información puede ser una pequeña cantidad de datos indicando que un "nuevo email" ha llegado notificándole a la aplicación principal que se debe realizar una sincronización en segundo plano, o podría ser un mensaje que puede contener como máximo un payload de 4kb de Data. El servicio GCM de Google maneja todos los aspectos de encolamiento, envío y recepción a los dispositivos Android correspondientes.

Las principales características del servicio GCM de Google son las Siguientes:

[3] Los Servidores Clientes son los servidores intermediarios entre la aplicación Android y el GCM.

- Permite a las aplicaciones/servidores clientes enviar mensajes a sus aplicaciones Android
- Utilizando el GCM Cloud Connection Server, se pueden recibir mensajes desde los dispositivos
- Una Aplicación en un dispositivo Android, no debe estar ejecutándose para recibir los mensajes. El Sistema Iniciará la aplicación via Intent Broadcast[4] cuando un mensaje llegue, siempre y cuando la aplicación esté correctamente configurada.
- No proporciona ninguna interfaz "prefabricada" para manejar/manipular los mensajes recibidos. GCM Simplemente pasa el mensaje "Crudo" directamente a la aplicación Android, lo que le da al desarrollador completo control a la hora de manejarlo.
- Requiere dispositivos con una versión de Android mayor a la 2.2 y que posean el Google Play Store instalado
- Utiliza una conexión existente a los servicios de Google. Para dispositivos menores a 3.0 es necesario poseer una cuenta de Google. Esto ya no es necesario para Dispositivos que ejecuten una versión de Android Superior a la 4.0.4

Conceptos Claves

En las siguientes tablas se resumen los términos caves y conceptos involucrados en el GCM.
Estos conceptos se dividen en categorías: [8]

- **Componentes** – las entidades que tienen un papel principal en GCM (Tabla 1).
- **Credenciales** – Los IDs y tokens que son utilizados en el GCM para garantizar que todas las partes han sido autenticadas, y que el mensaje va al lugar correcto (Tabla 2).

[4] El Intent Broadcast es un tipo de mensaje enviado por una aplicación o el Sistema Operativo para que sea capturado por algún BroadcastReceiver configurado.

Componente	
Servidores de conexión GCM	Los servidores de Google involucrados en enviar los mensajes entre el servidor cliente y enviarlos a los dispositivos Android.
Aplicación de Cliente	Una aplicación cliente configurada con el GCM que se comunicará con la aplicación del servidor cliente.
Aplicación de Servidor Cliente	Aplicación de servidor escrita por el desarrollador. Esta aplicación envía datos a los dispositivos finales a través del GCM.

Tabla 1 - Componentes GCM.

Credenciales	
Sender ID	Un número de proyecto que se obtiene del la consola de API. El Sender ID es utilizado en el proceso de registro para identificar a la Aplicación de Servidor Cliente que tienen permitidos enviar mensajes a los dispositivos finales.
Sender Auth Token	Una clave para el uso del API que es almacenada en la Aplicación de Servidor Cliente. Esta clave autoriza a la aplicación del servidor a utilizar los servicios de Google.
Application ID	La aplicación Android que está registrada para recibir mensajes. La aplicación Android es identificada por el nombre de paquete en el archivo Manifest. Esto asegura que el mensaje irá a la aplicación Correcta.
Registration ID	Un ID proporcionado por los servidores GCM a la aplicación Android para que esta pueda recibir los mensajes. Una vez que la aplicación tenga el Registration ID debe enviarlo a la aplicación de servidor cliente, que lo utilizará para identificar cada dispositivo registrado para recibir mensajes. En otras palabras, un Registration ID está atado a una aplicación Android que se ejecuta en un dispositivo.

Tabla 2 - Credenciales GCM.

Arquitectura:

Una implementación del GCM incluye una conexión a los servidores de Google. Un servidor Cliente que interactúa con el servidor GCM, y con una aplicación cliente con el servicio GCM ejecutándose en el dispositivo Android. La Figura 11 muestra un diagrama del Servidor Cliente comunicándose con una aplicación cliente Android a través del GCM.

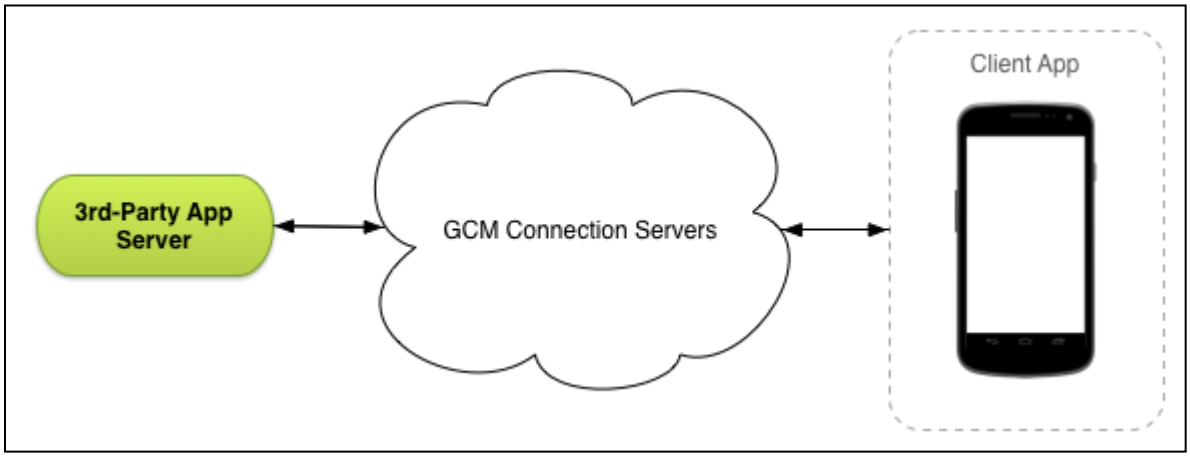


Figura 11 Arquitectura GCM.

Los componentes interactúan de la siguiente manera:

- Los servidores de Google (GCM Connection Server) toman el mensaje del Servidor Cliente y envían este mensaje al dispositivo Android.
- El Servidor Cliente es un componente que debe ser implementado para que trabaje con los servidores de Google. El Servidor Cliente envía los mensajes a los servidores de Google, y este se encarga de entregarlo a los dispositivos finales
- El Cliente Android recibe los mensajes desde el GCM. Para recibir los mensajes la aplicación del cliente debe estar registrada con Google y debe tener el Registration ID.

Workflow

A continuación se detalla el flujo de trabajo de una aplicación basada en push GCM.

Habilitar el servicio GCM:

La aplicación Android en el dispositivo se registra para recibir mensajes. La primera vez que la aplicación necesita enviar un mensaje, hace un llamado al método *register()* , este método retorna el Registration ID. La aplicación Android debe almacenar este ID para usarlo posteriormente.

Enviar Mensajes:

La secuencia de eventos para enviar un mensaje es la siguiente:

1. La aplicación del servidor envía un mensaje a los servidores de Google.
2. Google encola y almacena el mensaje en caso de que el dispositivo esté desconectado.
3. Cuando el dispositivo se conecta, Google envía el mensaje al dispositivo.
4. En el dispositivo, el sistema envía el mensaje vía Broadcast a la aplicación específica. Esto inicia la aplicación que no necesita estar activa ni ejecutándose para poder recibir los mensajes.
5. La aplicación Procesa el mensaje.

La aplicación puede De suscribirse si no quiere seguir recibiendo mensajes.

Recibir Mensajes:

La secuencia de eventos cuando la aplicación recibe un mensaje es la siguiente

1. El sistema recibe el mensaje entrante y extra el par valor/clave del payload.
2. El sistema pasa el par valor/clave a la aplicación indicada en un *Intent* del tipo *com.google.android.c2dm.intent.RECEIVE* como un set de extras.
3. La aplicación Extrae los extras del *Intent con.google.android.c2m.intent.RECEIVE* y procesa los datos.

2.6 Otras Alternativas

PushWoosh[5]

Pushwoosh es un servicio de notificaciones multiplataforma gratuito y pago, que permite a los desarrolladores, representantes de marketing y dueños de productos estar en contacto con los usuarios de sus aplicaciones, promover productos, aplicaciones, ventas, y mantener un tracking de campañas con notificaciones. Pushwoosh provee acceso instantáneo al servicio, se encargan de la carga y permite que los desarrolladores se enfoquen en crear los productos.

PushWoosh ofrece librerías que se encarga de recibir y mostrar los mensajes enviados en la barra de notificaciones. Ahorrando así todo este trabajo al desarrollador.

A pesar de que la librería funciona bastante bien internamente utiliza el servicio GCM. Por lo que se debe tener una cuenta de Google y se deben crear los token de autenticación para que los servidores de PushWoosh puedan utilizar el servicio.

Pushover[6]

Es un servicio pago para recibir notificaciones push en tablets o teléfonos de una variedad de fuentes, todo esto a través de una aplicación cliente que debe ser descargada e instalada por el usuario. Esta empresa proporciona Apis http para para el envío de mensajes a los dispositivos finales. Pushover sólo funciona con sus propias aplicaciones clientes para cada plataforma, lo que representa una gran desventaja a la hora de querer realizar aplicaciones propias con funcionalidad push utilizando esta API.

Urban Airship[7]

Urban Airship es una empresa que ofrece soluciones tecnológicas para tecnología móvil. Al igual que PushWoosh ofrece Apis, librerías y una interfaz web para el envío de mensajes push

[5] https://www.pushwoosh.com/

[6] https://pushover.net/

[7] http://urbanairship.com/

a múltiples plataformas, desafortunadamente depende de los servicios propios de Google (GCM) para el arribo de estos mensajes.

Parse[8]

Servicio pago que ofrece Apis y librerías para el envío de mensajes push a dispositivos Android (y otras plataformas) a través del uso de los servicios de Google GCM.

La ventaja de estas alternativas varían de acuerdo al fabricante algunas de estas librería tiene opciones muy completas que de seguro ahorrarán tiempo a la hora de desarrollo, sin embargo algunas son pagas, la mayoría es dependiente del servicio GCM de Google o tiene servicios y aplicaciones propias que permiten al desarrollador tener el control total del servicio de mensajería. Es por esto que en el capítulo siguiente, se presenta una alternativa para poder hacer uso de un sistema de mensajería push en donde el desarrollador podrá tener el control absoluto de este servicio.

[8] https://www.parse.com

Capítulo 3: Diseño y Desarrollo de la Propuesta

En este capítulo se plantea el diseño, el detalle y funcionamiento de la solución propuesta, los lenguajes, las tecnologías utilizadas y el flujo entre todas las partes para el correcto funcionamiento del sistema.

La metodología seleccionada para la elaboración de este proyecto, es realizar el desarrollo utilizando la metodología Scrum, que es un modelo de desarrollo ágil que se caracteriza por realizar entregas parciales y regulares del producto final, consiguiendo así un desarrollo incremental, en lugar de la planificación y ejecución completa del producto.

3.1 Diseño de la Solución

Para dar solución al problema, se debe tener en cuenta las 2 entidades que participan en el proceso de comunicación Push. El Cliente y el Servidor, Ambas partes deber ser configurables y fácilmente integrables para la creación de cualquier sistema que quiera utilizar esta forma de comunicación.

La Solución propuesta consiste en el desarrollo de dos librerías que podrán ser utilizadas en cualquier tipo de proyecto Android (cliente) y Windows (servidor) para la implementación de un sistema de mensajería Push.

Estas librerías deben ser utilizadas por cada una de las entidades que interactúan en el proceso de comunicación Push (Cliente, servidor). El cliente o dispositivo Android y el Servidor que se ejecutará en un equipo Windows.

Los requisitos se detallan a continuación para cada una de las partes involucradas en el proceso.

3.1.1 Cliente

La Librería a ser utilizada por el cliente debe ser capaz de realizar la conexión inicial hacia el servidor siempre que ambas entidades se encuentren dentro de la misma red. Esta conexión debe permanecer siempre abierta y activa chequeando constantemente que la conexión no haya finalizado. En caso de que la conexión haya finalizado repentinamente se debe realizar un proceso de reconexión hasta que la misma haya sido restablecida exitosamente. Como la

librería cliente es una librería que será utilizada en dispositivos móviles se debe establecer la lógica para que no se hagan intentos de conexión si el equipo no tiene señal de datos y tiene la conexión WiFi desactivada. La Interfaz por la donde se realizará la comunicación de datos serán las interfaces de Datos y WiFi del equipo. La aplicación cliente debe recibir los mensajes vía Broadcast a través de una clase BroadcastReceiver.

3.1.2 Servidor

La librería a ser utilizada por el servidor debe ser capaz recibir, procesar y mantener la conexión activa que es realizada por el cliente. Al mismo tiempo el servidor debe seguir esperando por nuevas conexiones entrantes y debe poder enviar mensajes a cada uno de los clientes que se encuentren conectados a él. Esta comunicación debe ser bidireccional. En caso que se intente enviar un mensaje a un equipo y este se encuentre desconectado el servidor debe almacenar este mensaje en la Base de Datos con un estatus de “No Enviado” y cuando el cliente se conecte nuevamente debe recibir inmediatamente los mensajes almacenados que llegaron durante su ausencia.

Debe existir una opción para que el mensaje se almacene o no. Es decir, el mensaje puede ser configurado para que llegue sólo a los equipos que se encuentran conectados actualmente o para que llegue a todos los equipos registrados en el sistema.

Adicionalmente el servidor debe tener la opción de escuchar conexiones por un puerto independiente. Esta conexión será utilizada por el administrador para realizar las consultas o enviar mensajes a los dispositivos.

Se deben crear métodos virtuales que podrán ser sobrescritos por el programador para que a la hora de utilizar la librería se pueda controlar el comportamiento al recibir o enviar mensajes hacia o desde los dispositivos.

La librería que se desarrolle para el servidor debe tener acceso a una Base de Datos que se encargará de almacenar los dispositivos registrados así como los mensajes enviados y su estatus.

Ambas librerías deben definir su propio protocolo de comunicación para manejar los mensajes de confirmación, los mensajes de control de la aplicación y los mensajes enviados por los dispositivos.

3.2 Modelo de caso de Uso

A continuación en la Figura 12 se muestra el diagrama de caso de uso.

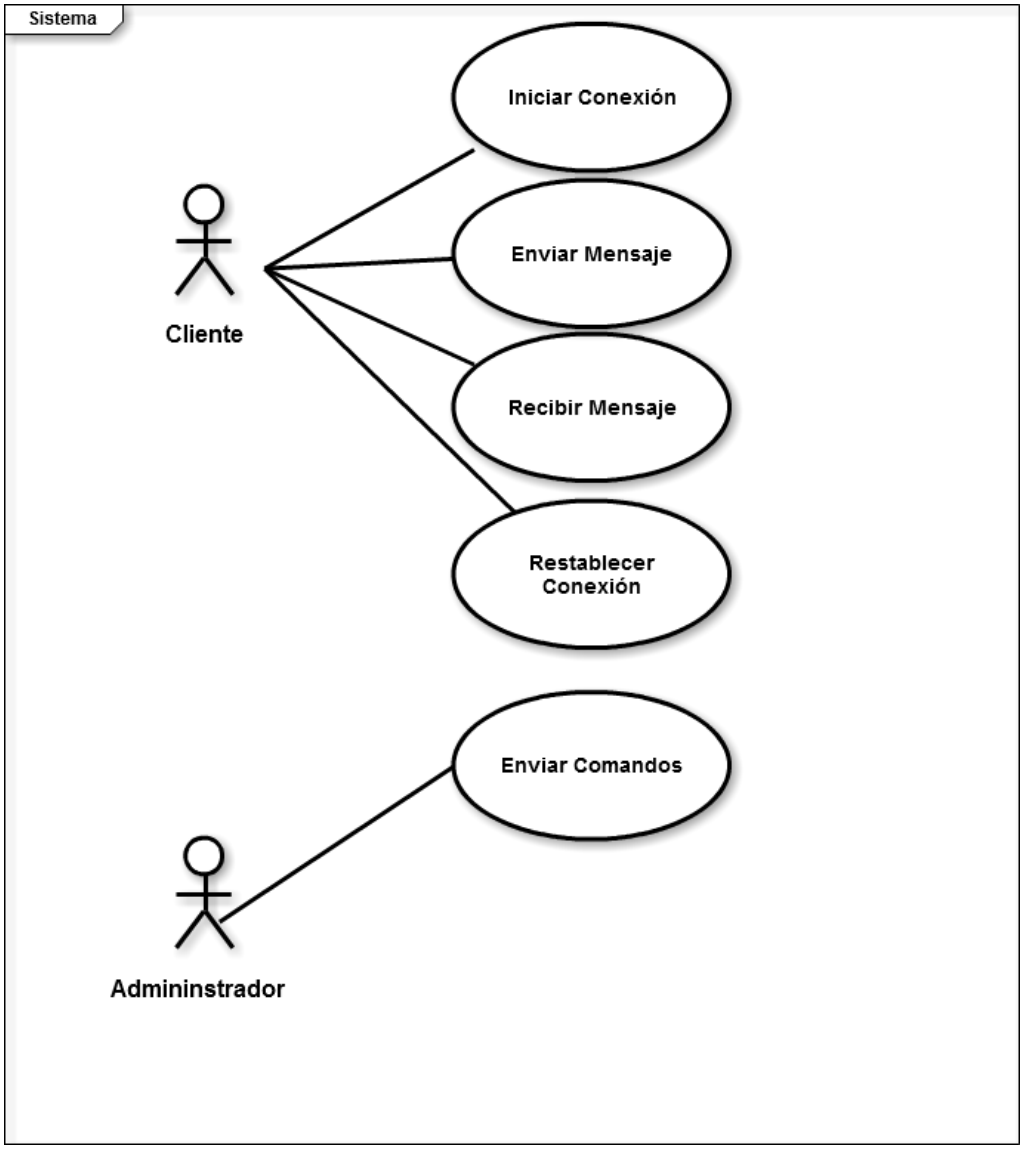


Figura 12 Modelo de Caso de Uso Cliente Push

- **Iniciar Conexión**: caso de uso mediante el cual la aplicación cliente realiza la conexión inicial hacia el servidor push.
- **Enviar Mensaje**: caso de uso mediante el cual la aplicación cliente realiza el envío de mensaje hacia el servidor push.
- **Recibir Mensaje**: caso de uso mediante el cual la aplicación cliente permanece a la escucha de los mensajes que puedan llegar desde el servido.
- **Restablecer Conexión**: caso de uso mediante el cual la aplicación cliente debe intentar restablecer la conexión hacia el servidor push si esta se ha perdido.

3.3 Diagrama de secuencia.

En la Figura 13 muestra el flujo completo entre el cliente y el servidor.

3.4 Lenguajes de Programación a Utilizar

Los lenguajes de programación que serán utilizados para el desarrollo de las librerías son:

Visual C#

Es un lenguaje de programación moderno de alto nivel, múltiples paradigmas y de uso general para crear aplicaciones con Visual Studio NET Framework. Este lenguaje se diseñó para que fuera simple, poderoso, seguro y orientado a Objetos.

Java

Es un lenguaje de programación de propósito general, concurrente, orientado a objetos y basado en clases que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. Su intención es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo (conocido en inglés como WORA, o "write once, run anywhere"), lo que quiere decir que el código que es ejecutado en una plataforma no tiene que ser recompilado para correr en otra.

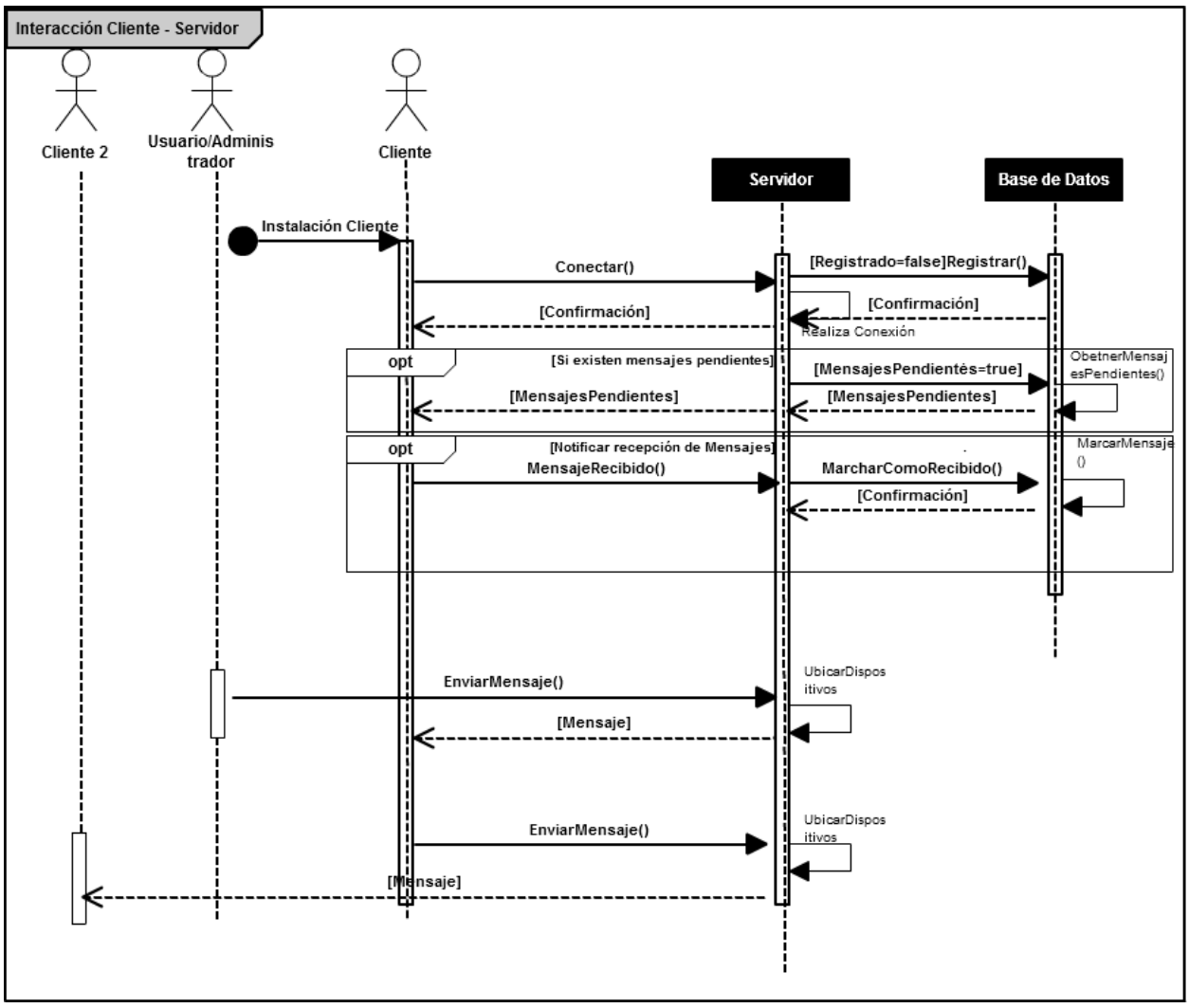


Figura 13 Diagrama de Secuencias entre el cliente y el servidor

3.5 Persistencia de Datos

SQLServer

Es un sistema para la gestión de bases de datos producido por Microsoft basado en el modelo relacional. Como base de datos es un producto de software que tiene como funcionalidad primaria almacenar y obtener data requerida por otras aplicaciones de software, ya sea que estén en la misma computadora o a través de la red (Incluyendo Internet). Sus lenguajes para consultas son T-SQL y ANSI SQL.

3.6 Servicio Windows (Servidor)

Diseño librería servidor (Push.dll)

Se creará una clase abstracta que podrá ser utilizada para el desarrollo de servicios o aplicaciones que corran bajo el Sistema Operativo Windows. Cualquier aplicación que se cree debe poder heredar de esta clase que será la encargada de manejar los métodos que se encargaran de escuchar y mantener las conexiones existentes en el servidor. Los mensajes de texto serán enviado utilizando la codificación UTF-8 por lo que cada byte será de 8 bits. El tamaño máximo recomendado para el envío de datos es de 400KBytes. En la Figura 14 se tiene una representación de la Clase push. En la Tabla 3 se listan y explican los atributos de la clase y en la Tabla 4 muestran los métodos utilizados por la clase. Los Métodos públicos pueden ser accedidos por el usuario y los métodos virtuales son los utilizados para agregar la lógica al recibir mensajes de los clientes. El código 1 contiene los atributos y métodos de la clase push en c#.

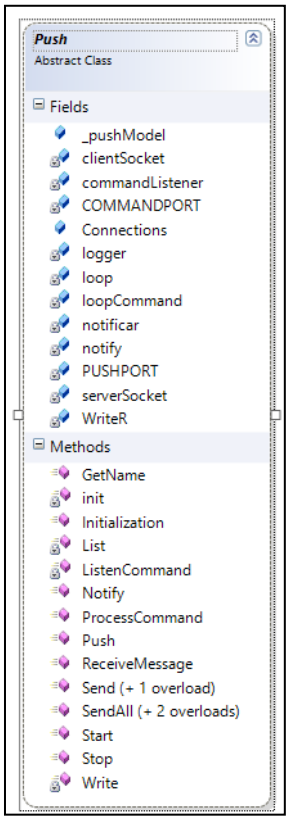


Figura 14 Clase Push

```
      public abstract class Push
    {

        #region Attributes
        private static int PUSHPORT = 8888, COMMANDPORT = 8899;
        public static Dictionary<string, Dispositivo> Connections;
        public static Model.BLL.Push _pushModel;
        Thread WriteR;
        Thread notify;
        Thread commandListener;
        TcpListener serverSocket;
        TcpClient clientSocket;
        private bool loop = true, loopCommand = true, notificar = false;
        #endregion

        #region Private
        void Write();
        int List();
        void ListenCommand();
        void init();
        #endregion

        #region Virtuals
        public virtual void Notify();
        public virtual string ProcessCommand(string trama);
        public virtual void ReceiveMessage(long msjId, byte[] bytesFrom, String id);
        public virtual void Initialization();
        public virtual string GetName(string trama);
        #endregion

        #region Publics
        public Push();
        public void Start();
        public void Stop();


        public int SendAll(String device, long msjId, string msj);
        public int SendAll(String device, long msjId, string msj, bool confirmation);
        public int SendAll(String devId, long msjId, string msj, bool confirmation, String but);
        public void Send(String devIdOrig, long msjId, List<String> devIdDestList, string msj, bool
confirmation);
        public void Send(String devIdOrig, long msjId, List<String> devIdDestList, string msj);
        #endregion
    }
```

Código 1 Clase Push (C#)

Atributos clase Device:

Atributo	Descripción
`Model.BLL.Push _pushModel;`	Variable para instanciar la clase que se contiene los métodos que se encargarán de la consulta y persistencia de datos.
`TcpClient clientSocket;`	Cliente TCP que provee métodos para conectar, enviar y recibir datos a través de la red. Atributo utilizado para recibir los mensajes recibidos por el cliente.
`Thread commandListener`	Hilo que se encargará de escuchar las peticiones que no sean de los clientes (Conexiones – del administrador).
`int COMMANDPORT`	Puerto a ser utilizado para recibir las conexiones por parte del administrador. Puerto por defecto 8899.
`Dictionary<string, Dispositivo> Connections`	Estructura que contendrá los dispositivos conectados utilizando como índice para su acceso el ID del mismo. El Id de los dispositivos es generado automáticamente por la librería push. Es una combinación del IMEI(en caso de que exita), la MAC address del equipo, y un ID “único” que provee el framework Android.
`Thread escribirR`	Método utilizado para las pruebas.
`bool loop`	Booleano que indicará si se debe mantener el ciclo que escucha por una conexión entrante para los dispositivos clientes o no.
`bool loopCommand`	Booleano que indicará si se debe mantener el ciclo que escucha por una conexión entrante para los administradores o no.
`bool notificar`	Variable configurable que indica si el servidor es el que debe chequear constantemente por el estado de los clientes conectados.
`Thread notify`	Hilo que se encargará de ejecutar el método

Atributo	Descripción
	encargado de realizar la verificación de estatus a los clientes
`int PUSHPORT`	Puerto a ser utilizado para recibir las conexiones por parte de los clientes. Puerto por defecto 8888.
`TcpListener serverSocket`	Clase que proporciona los métodos para escuchar las conexiones entrantes utilizando la clase `TcpClient`.

Tabla 3 Atributos de la clase Push

Métodos clase Device:

Método	Descripción
Privados	
`void init()`	Método que inicializa los parámetros de conexión y configuración de la aplicación (puertos, notificaciones, etc).
`void ListenCommand()`	Método que ejecutará el hilo commandListener y es el encargado de escuchar las conexiones o los comandos que lleguen externamente (No de los clientes).
`void escribir()`	Método que escucha por comandos introducidos vía consola (pruebas).
`int Listar()`	Lista los dispositivos cliente que se han conectado y su estado actual.
Públicos	
`public Push()`	Método constructor de la clase.
`public void Start()`	Método que escucha Formalmente la conexión con los dispositivos clientes. El método se encarga de crear el nuevo Dispositivo pasando como uno de los parámetros la conexión

Método	Descripción
	existente (Ver clase Dispositivo).
`public void Stop()`	Se encarga de cerrar las conexiones existentes.

`public int SendAll(String devId, long msjId, string msj, bool confirmation, String but);`
 	
`public int SendAll(String device, long msjId, string msj, bool confirmation);`
 	
`public int SendAll(String device, long msjId, string msj);`
	Se encarga de enviar un mensaje a todos los dispositivos.
	
	Parámetros:
	
	devId: Id del dispositivo origen.
	
	msjId: Id local al dispositivo Android que envió el mensaje.
	
	msj: Contenido del mensaje.
	
	confirmation: booleano que indica si el mensaje debe o no ser encolado.
	
	but: Id del dispositivo que debe ser excluido el envío del mensaje.(para cuando se envíen mensajes a todos menos al que envía el mensaje)
	
	Retorna la cantidad de dispositivos a la que se le envió el mensaje.

`public void Send(String devIdOrig, long msjId, List<String> devIdDestList, string msj, bool confirmation);`
 	
`public void Send(String devIdOrig, long msjId, List<String> devIdDestList, string msj);`
	Método que se encarga de enviar un mensaje a una lista especifica de dispositivos.
	
	Parámetros:
	
	devIdOrig: Id del dispositivo origen.
	
	msjId: Id local al dispositivo Android que envió el mensaje.
	
	msj: Contenido del mensaje.
	
	confirmation: booleano que indica si el mensaje debe o no ser encolado.
	
	devIdDestList: Lista con los Id de los dispositivos a los cuales se le enviará el mensaje.

Método	Descripción
Virtuales (Métodos programables por el desarrollador)	
`public virtual void Notify();`	Método que ejecutará el hilo *notify* y está destinado a enviar notificaciones mensajes periódicas a los clientes.
`public virtual string ProcessCommand(string trama);`	Método que procesará los comandos que se reciban externamente que pueden ir destinado o no a los clientes.
`public virtual void ReceiveMessage(long msjId, byte[] bytesFrom, String id);`	Método que recibirá el mensaje de los clientes. En este método se debe agregar la lógica correspondiente a las acciones que deben ser tomadas por parte del servidor para procesar los mensajes recibidos.
`public virtual void Initialization();`	Método que se ejecutará al iniciar la aplicación. Destinado para que se inicialicen todas las variables que serán utilizadas.
`Public virtual string GetName(string trama)`	Método que puede ser sobrescrito por el programador, utilizado para obtener el Id del dispositivo cliente.
`public void EnviarMensajeDB(Model.push_Mensaje msj, bool encolar);`	Método encargado de validar que el dispositivo esté conectado para enviar el mensaje, y de almacenar el mensaje en BD en caso de que el cliente no se encuentre conectado. (Si el mensaje es nuevo siempre se almacenará en base de datos. Parámetros: **msj:** Objeto de tipo Model.push_Mensaje que contendrá información asociada el mensaje. **encolar:** variable que indica si se debe o no encolar el mensaje que se desea enviar.

Tabla 4 Métodos de la clase Push

Cada cliente se representará con la clase Device **(Device.cs)** que se encargará de mantener la conexión con cada uno de los equipos conectados y escuchar (recibir) y escribir (enviar) los mensajes entrantes y salientes desde el dispositivo cliente hasta el servidor y viceversa. En la Figura 15 se tiene una representación de la clase Device. En la Tabla 5 se listan y explican los atributos de la clase y en la Tabla 6 muestran los métodos utilizados por la clase El código 1 contiene los atributos y métodos de la clase Device en c#.

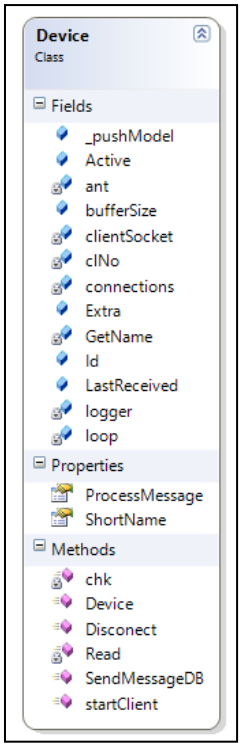


Figura 15 Clase Dispositivo.

```
public class Device
{
    #region Attributes
    static Dictionary<string, Device> connections;
    TcpClient clientSocket;
    string clNo;
    public string Id;
    public bool Active;
    string ant;
    int loop;
    public DateTime LastReceived;
    public static Model.BLL.Push _pushModel;
    public static int bufferSize = 20025;
    public object Extra;
    public string ShortName;
    public Action<long, byte[], string> ProcessMessage;
    GetId GetName;
    #endregion
```

```
        #region Private
        private void Read();
        private void chk();
        #endregion

        #region Public
        public Device(Dictionary<string, Device> disp, TcpClient inClientSocket, string clineNo,
Action<long, byte[], string> accion, GetId funcionId);
        public void startClient(TcpClient inClientSocket, string clineNo);
        public void Disconect();
        public void SendMessageDB(Model.push_Message Message, bool encolar);
        #endregion
    }
```

Código 2 Clase Device (C#)

Atributos clase Device:

Atributo	Descripción
`Dictionary<string, Dispositivo> conexiones;`	Referencia al Diccionario externo que contendrá la lista de todos los dispositivos conectados utilizando como clave el Id del dispositivo. Utilizado al momento de iniciar la conexión del dispositivo para saber si ya existe en la lista, en caso contrario el dispositivo se agrega a si mismo a la lista de dispositivos.
`TcpClient clientSocket`	Cliente TCP encargado de mantener la conexión abierta con el cliente final, una vez activo el cliente TCP se mantiene escuchando por nuevos mensajes, a través de él se realiza el envío los mensajes al dispositivo cliente.
`string clNo;`	String Utilizado para Identifica numéricamente al Dispositivo dentro de la lista de Dispositivos.
`Public string Id;`	String que contiene el Nombre (Id) del dispositivo Actualmente conectado.
`public bool Active;`	Booleano que indica si el dispositivo se encuentra Activo(Conectado) o Inactivo(Desconectado)
`string ant;`	String que almacena el último mensaje recibido. Se utiliza para detectar conexiones abruptas que generan un ciclo infinito y hacen creer que

Atributo	Descripción
	siempre se recibe el mismo valor.
`int loop;`	De la mano con la variable string ant, indica si se ha caído dentro de un ciclo infnito, esta variable se incrementará cada vez que se reciba un valor idéntico al anterior, Cuando el valor es el mismo 10 veces se asume que se ha perdido la conexión y se cierra(desconecta el dispositivo).
`public DateTime LastReceived;`	Marca la Fecha y hora del último Mensaje recibido.
`public static Model.BLL.Push _pushModel;`	Objeto que contiene las funciones y métodos que se encargaran de todo lo relacionado a la persistencia de datos.
`public static int bufferSize;`	Variable que define el tamaño del buffer que manejará el cliente TCP para la recepción de mensajes.
`public Action<byte[],string> ProcesarMensaje { get; private set; }`	Variable de Tipo Action<T1,T2> que encapsula un método con 2 parámetros. Esta función se define en la clase Push y es la encargada de procesar el mensaje recibido por el cliente siempre y cuando el mensaje no sea alguno de los mensajes de control que son utilizados internamente por la librería.
`GetId GetName;`	Variable de tipo public delegate string GetId(string trama); esta function se encarga de retornar el Id del dispositivo data una trama recibida. Esta función es definida en la clase Push.

Tabla 5 Atributos de la clase Dispositivo

Métodos clase Device:

Método	Descripción
Privados	

Método	Descripción
`private void Read()`	Método que inicia Formalmente la conexión con los dispositivos conectados. Se encarga de leer los datos enviados por el dispositivo cliente y se encarga de interpretar dicho mensaje ya sea un mensaje interno o un mensaje enviado por el programa cliente. El método se encarga de agregar al Dispositivo a la lista de dispositivos en caso de que no exista, se encarga de buscar todos mensajes en Base de datos que están pendientes para el dispositivo, enviarlos utilizando el método EnviarMensajeDB(), y se encarga de recibir las confirmaciones en la recepción de los mensajes enviados.
`private void chk() [Opcional]`	Método que se encargará de enviar un mensaje de "ping" para verificar si el equipo está actualmente conectado.
`void startClient(TcpClient inClientSocket, string clineNo);`	Método que se encarga de asignar los valores recibidos por el constructor a las variables internas, así como la inicialización de variables de control. En este método se inicia la ejecución del método Leer() que es el encargado de iniciar la conexión, y escuchar los mensajes recibidos.
Públicos	
`public Device(Dictionary<string, Dispositivo> disp, TcpClient inClientSocket, string clineNo, Action<byte[], string> accion, GetId funcionId)`	Constructor para instanciar el objeto e iniciarlo llamando al método startClient(inClientSocket, clineNo); Parámetros: **disp**: Objeto de tipo Dictionary<string, Dispositivo> que se encuentra en la clase Push y que contiene todos los dispositivos que se están conectados o que alguna vez lo estuvieron, esta referencia es asignada a la

Método	Descripción
	variable conexiones de la clase Dispositivo. **inClientSocket:**Cliente TCP que contiene la conexión con el cliente, este parámetro es asignado a la variable clientSocket de la clase Dispositivo. **clineNo:** String que contiene el identificador numeric de la conexión actual, este parámetro es asignado a la variable clNo de la clase Dispositivo. **acción:** parámetro de tipo Action<T1,T1> que encapsula un método que tiene 2 parámetros, este parámetro es asignado a la variable ProcesarMensaje de la clase Dispositivo. Esta función se define en la clase Push y es la encargada de procesar el mensaje recibido por el cliente siempre y cuando el mensaje no sea alguno de los mensajes de control que son utilizados internamente por la librería. **funcionId:** parámetro de tipo delegate string GetId(string trama); este parámetro es asignado a la variable GetName de la librería Dispositivo
`public void EnviarMensajeDB(Model.push_Mensaje msj, bool encolar);`	Método encargado de validar que el dispositivo esté conectado para enviar el mensaje, y de almacenar el mensaje en BD en caso de que el cliente no se encuentre conectado. (Si el mensaje es nuevo siempre se almacenará en base de datos. Parámetros: **msj:** Objeto de tipo Model.push_Mensaje que contendrá información asociada el mensaje. **encolar:** variable que indica si se debe o no encolar el mensaje que se desea enviar.

Método	Descripción
`public void Disconect()`	Método que se encarga de liberar la conexión con el dispositivo Actual.

Tabla 6 Métodos de la clase Dispositivo

Envío y recepción de datos

La librería Push implementa un protocolo interno para la interpretación de los mensajes enviados/recibidos y los mensajes de control. Todo lo que se transmita vía TCP será texto utilizando codificación UTF-8 para diferenciar y delimitar mensajes de control y los mensajes enviados/recibidos se utilizarán una serie de bytes de control que estarán definidos tanto en el cliente como en el servidor. El significado de estos bytes está definido en la Tabla 7.

```
namespace Push
{
    public class Common
    {
        public static byte _beatByte = 17;
        public static byte _endBye = 23;
        public static byte _idByte = 18;
        public static byte _startByte = 15;
        public static byte _headermessageByte = 1;
        public static byte _messageByte = 2;
        public static byte _messageACKByte = 6;
        public static byte _messageStartByte = 77;//Letra M
    }
}
```

Código 3 Clase Common con bytes de control (C#)

Byte	Significado
_beatByte(17)	Indica si el mensaje recibido es un beat del cliente.
_endBye(23)	Indica la finalización del mensaje actual.
_idByte(18)	Indica que el mensaje es el Id del dispositivo.
_startByte(15)	Byte inicial para el inicio de la conexión

Byte	Significado
	con el cliente.
_headermessageByte(1)	Byte para indicar que el mensaje contiene un id local (cliente).
_messageByte(2)	Indica el inicio del mensaje enviado por el cliente.
_messageACKByte(6)	Byte enviado al cliente seguido del Id del mensaje para confirmar la.
_messageStartByte(77)	Indica el inicio del mensaje enviado hacia el cliente.

Tabla 7 Definición de bytes para el envío y recepción de mensaje.

Modelo

La librería Push interactúa con la librería Model que es la encargada del acceso a Base de Datos y la persistencia de datos. Los métodos de esta librería son utilizadas sólo por la clase push pero contiene la definición de la clase Message que puede ser utilizada por otras aplicaciones en .NET externa para comunicarse con el servicio push.

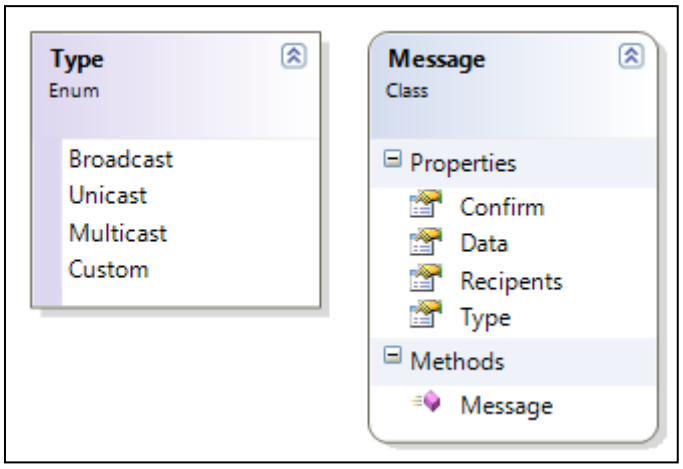


Figura 16 Clase Message .

En la Figura 16 se observa la clase *Message* diseñada para que actúe como el contenedor de los mensajes que se reciben vía Comando. Esta clase consta de la variable booleana *Confirm* para indicar si el mensaje a enviar requiere confirmación o no, la clase Data

de tipo *object* para que reciba cualquier objeto (Ejemplo el *String* con el mensaje), *Recipents* una lista de *String* que contiene el o los dispositivos destinos a los cuales se les hará llegar el mensaje, y un tipo enumerado que indica a qué grupo será enviado el mensaje. *Broadcast* indica que el mensaje será enviado a todos los dispositivos conectado, por lo que no es necesario enviar Ids en el campo *Recipents*, *Unicast* indica que el mensaje será enviado a un solo dispositivo cuyo Id debe estar en la lista *Recipents* (primera y única posición), *Multicast*, el mensaje será enviado sólo a los Id que se especifiquen en el campo *Recipents* y *Custom* para realizar cualquier otra cosa deseada por el usuario.

Configuración

Al utilizar la librería push se debe incluir en el archivo de configuración el *connectionString* que contiene los datos de conexión a la Base de datos en donde se almacenan los mensajes push el *connectionString* tiene el nombre `DBConnection`. Adicionalmente en la sección *appSetting* se deben incluir el siguiente conjunto de pares `PushPort`, `CommandPort`, `Notify` que representan el puerto por donde se recibirán las conexiones push(clientes) el puerto para recibir los comandos(Admin) y un booleano indicando si se debe ejecutar el método notify definido anteriormente. En la Figura 17 se muestran estos parámetros en el archivo de configuración.

```
1  <?xml version="1.0"?>
2  <configuration>
3    <connectionStrings>
4      <add name="DBConnection" connectionString="data source=invasor,1802;Initial
5    </connectionStrings>
6    <appSettings>
7      <add key="PushPort" value="8890"/>
8      <add key="CommandPort" value="8891"/>
9      <add key="Notify" value="false"/>
10   </appSettings>
11   <startup>
12     <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/>
13   </startup>
14 </configuration>
15
```

Figura 17 Documento de configuración de la aplicación.

Manejo de errores

La librería es capaz de manejar errores relevantes al envío de mensajes hacia los dispositivos. Si por algún motivo existe una conexión muerta hacia un dispositivo la librería almacenará el mensaje hasta que el dispositivo vuelva a conectarse. En el momento cuando el dispositivo realice la primera conexión le serán entregado lo mensajes que fueron enviado a él durante su ausencia.

3.7 Servicio Android (Cliente)

Diseño librería cliente (PushService)

Se creará una librería que podrá ser utilizada para cualquier proyecto de tipo Android utilizando eclipse. Esta Liberia deber crearse como un Servicio Android que mantendrá una conexión constante hacia el servidor, implementar métodos para detectar la desconexión e intentar la reconexión. Debe ser capaz de recibir y procesar los mensajes recibidos. Así como enviar mensajes al servidor push en caso de ser necesario. La librería debe poder almacenar internamente los mensajes que no hayan podido ser enviados por problemas de conexión y enviarlos cuando la conexión se encuentre disponible. La librería también debe evitar las reconexiones innecesarias cuando el dispositivo se encuentre con el WiFi desconectado y la red de datos deshabilitada.

Clase TCPClient

Clase principal que se encarga de crear abrir y conectar el socket que realiza la conexión al servidor. Esta clase contiene los métodos para enviar/recibir mensajes, verificar el estado del enlace y los parámetros necesarios para realizar la conexión al servidor. El Código 4 contiene los atributos y métodos de la clase, la Tabla 8 contiene la descripción de los atributos, la Tabla 9 contiene la descripción de los métodos y la Tabla 10 contiene la descripción de la interfaz que se utiliza para la recepción de mensajes.

```
public class TCPClient {
    private String serverMessage;
    public static String SERVERIP ="192.168.2.4";
    public static  int SERVERPORT = 8888;
    public int ConectionTimeout=15000;
    final long MAX_WAIT_INTERVAL=18000000;
    private OnMessageReceived mMessageListener = null;
    private boolean mRun = false;
```

```
    Socket socket;
    PrintWriter out;
    BufferedReader in;
    long waitTime;
    Context context;
    public TCPClient(Context ctxt,OnMessageReceived listener);
    public  boolean sendMessage(String message);
    public boolean sendControlMessage(String message);
    private long getWaitTimeExp(int retryCount);
    public boolean beat();
    public void stopClient();
    public synchronized void run();
    public interface OnMessageReceived {
        public void messageReceived(String message);
    }
}
```

Código 4 Clase TCPClient (Java)

Atributos clase TCPClient:

Atributo	Descripción
`private String serverMessage`	String que contiene el mensaje recibido desde el servidor.
`public static String SERVERIP`	String que indica el nombre del host o la IP del servidor.
`public static  int SERVERPORT`	Entero que indica el puerto en el servidor al cual se va a realizar la conexión.
`public int ConectionTimeout`	Tiempo de espera en milisegundos para que el socket realice la conexión hacia el servido antes de arrojar un error de conexión por Timeout.
`final long MAX_WAIT_INTERVAL`	Tiempo de espera máximo en milisegundos definido para el intento de reconexión al servidor push.
`private OnMessageReceived mMessageListener`	Interfaz que debe ser implementada en el servicio y contiene definido el método que será llamado cuando arribe un nuevo mensaje.
`private boolean mRun`	Variable que indica si la conexión debe seguir escuchando o no.
`Socket socket;`	Socket para realizar la conexión al servidor.
`PrintWriter out;`	PrintWriter utilizado para escribir/enviar los mensajes al servidor.

Atributo	Descripción
`BufferedReader in;`	BufferReader para leer los mensajes que llegan del servidor a través del socket.
`long waitTime;`	Variable que será utilizada para asignar el tiempo de espera en milisegundos para los intentos de reconexión hacia el servidor.
`Context context;`	Variable de contexto Android.

Tabla 8 Atributos de la clase TCPClient

Métodos clase TCPClient:

Método	Descripción
`public TCPClient(Context ctxt,OnMessageReceived listener);`	Constructor de la clase.
`public boolean sendMessage(String message);`	Función que se encarga de enviar mensajes al servidor utilizando el PrinterWriter de la clase.
`public boolean sendControlMessage(String message);`	Función que se encarga de enviar mensajes al servidor utilizando el PrinterWriter de la clase y que será utilizada por el servicio para enviar los mensajes de control.
`private long getWaitTimeExp(int retryCount);`	Función que retorna el tiempo de espera en milisegundos en base la cantidad de reintentos de conexión hacia el servidor.
`public boolean beat();`	Función que se encarga de enviar mensajes de "Ping" al servidor para validar que aún no se ha perdido la conexión. si ha pasado un periodo de tiempo largo (80000ms) desde la recepción del último mensaje de confirmación la función retorna falso indicando que la conexión se ha perdido.
`public void stopClient();`	Función que cierra los sockets y termina formalmente la conexión hacia el servidor
`public synchronized void run();`	Función que inicia la conexión al servidor y se mantiene en un ciclo esperando (escuchando) por nuevos mensajes que provengan del

	servidor.

Tabla 9 Métodos de la clase TCPClient

Interfaces Clase TCPClient:

Interfaz	Descripción
`public interface OnMessageReceived`	Intefaz que contiene el método `public void messageReceived(String message);` y define que hacer cuando arribe un mensaje desde el servidor.

Tabla 10 Interfaces de la clase TCPClient

Servicio ConnectionService

Servicio encargado de ejecutar y mantener activa la conexión hacia el servidor push utilizando la clase *TCPClient.* Por tratarse de un servicio el mismo correrá en background sin necesidad de que la aplicación principal se encuentre abierta. El código 5 contiene la estructura en Java de la clase, la Tabla 11 contiene los atributos con su descripción y la Tabla 12 define los métodos de esta clase.

```
public class ConnectionService extends Service {

    public static int retries=0;
    public static long lastErrorconnection=0;
    public static boolean WakeLock=true;
    public static boolean disableBackOff=false;
    enum  Flag{Active,Created,Connecting,Beating}
    private static  TCPClient TcpClient;
    static AsyncTask<?, ?, ?> p;
    public static boolean Active=false,Created=false,Connecting=false,Beating=false;
    PowerManager pm;
    PowerManager.WakeLock wl;
    public static int TimerCheck=30;
    private static int APP_ID;
    private String INTENT_FILTER;
    static Context context;
    public static Calendar LastResponse;

    public  void init();
    public static  void  SetInit(Context ctxt,String host,int port,int appId,boolean wakeLock);

    private final IBinder myBinder;
    @Override
    public IBinder onBind(Intent arg0);
    @Override
    public boolean onUnbind(Intent intent);
    @Override
    public void onCreate();
    @Override
```

```
    public void onDestroy();
    @Override
    public int onStartCommand(Intent intent, int flags, int startId);


    private synchronized void Conectar();
    private synchronized void CambiarEstado(Flag f,boolean estado);
    public  class connectTask extends AsyncTask<String,String,TCPClient> {...}
    private void checkPending();

    Handler timer;
    final Runnable checker;

}
```

Código 5 Clase ConnectionService (Java)

Atributos ConnetionService:

Atributo	Descripción
`public static int retries`	Variable que almacena la cantidad de reintentos de conexión hacia el servidor en un intervalo de tiempo.
`public static long lastErrorconnection`	Variable que almacena el tiempo del último error en la conexión hacia el servidor.
`public static boolean WakeLock`	Variable que indica si se debe mantener encendido el procesador activo mientras el servicio esté corriendo.
`public static boolean disableBackOff`	Variable que indica si se debe ignorar la espera incremental al momento de realizar intentos de reconexión hacia el servidor.
`enum  Flag`	Variable de tipo enum con los valores `{Active,Created,Connecting,Beating}`par a definir el estado de la servicio.
`private static  TCPClient TcpClient`	Objeto TCPClient encargado del proceso de comunicación entre el cliente y el servidor.
`static AsyncTask<?, ?, ?> p`	Variable `AsyncTask` encargado de ejecutar la tarea `connectTask`.
`public static boolean Active,Created,Connecting,Beating;`	Variables de control definidas para conocer el estado del servicio.
`private final int TIMEOUT=30;`	Tiempo de chequeo en Segundos para verificar que la conexión siga activa.

Atributo	Descripción
`private static int APP_ID;`	Id de la Aplicación que está corriendo el servicio.
`private String INTENT_FILTER;`	String que identificará los mensajes que serán enviados entre el servicio y la aplicación final.
`static Context context;`	Contexto de la apliación actual.
`public static Calendar LastResponse;`	Variable que indica cuando fue la última vez que se recibió un mensaje del servidor.

Tabla 11 Atributos de la clase ConnectionService

Métodos ConnectionService:

Método	Descripción
`public void init();`	Método que se encarga de inicializar algunas variables de la clase. Es llamado desde el método `onCreate();`
`public static void SetInit(Context ctxt,String host,int port,int appId,boolean wakeLock);`	Método estático para inicializar las variables desde la aplicación cliente que estará utilizando la librería. Utilizando este método se especifica, la Ip o nombre host del servidor push, el puerto por donde estará escuchando el servidor, un Id(digo de aplicación) y un booleano indicando si se debe mantener activo el cpu del teléfono durante el uso del servicio.
`@Override` `public void onCreate();`	Método sobrecargado que se ejecutará al crear el servicio. Acá se llama al método `init();`
`@Override` `public void onDestroy();`	Método destructor del servicio, este método cierra las conexiones/sockets existentes y reinicia las variables.
`public static long SendMessage(String msj, boolean deliver);`	Función utilizado por la aplicación cliente para enviar mensajes hacia el servidor, el parámetro msj es el String con el mensaje que será enviado y adicionalmente se envía un booleano indicando si se desea recibir confirmación por parte del servidor que llegó

Método	Descripción
	el mensaje. En ese caso la función retornará el Id del mensaje utilizado internamente por la aplicación.
`@Override public int onStartCommand(Intent intent, int flags, int startId);`	Método que se ejecuta al iniciar el servicio. En este método se inicializa el timer para chequear la conexión con el servidor y se realiza el primer intento de conexión utilizando el método `Conectar();`
`private synchronized void Conectar();`	Método que ejecuta la tarea asíncrona `connectTask` que es la encargada de ejecutar el TCPClient.

Tabla 12 - Métodos de la clase ConnectionService

Conectividad

La librería contiene un *BroadcastReceiver* que recibirá del sistema operativo información sobre el cambio en la conectividad del dispositivo. Es decir esta Receiver detecta si el teléfono no posee conexión a internet (la red de wiFi y Datos está desactivada) y detiene el servicio de forma automática para evitar intentos de conexión que resultarían fallidos ya que hay acceso a la red. Así mismo cuando el *Receiver* detecte que la conexión está activa nuevamente este procederá a levantar nuevamente el servicio que se encargará de realizar la conexión con el servidor push.

Modelo y Base de Datos

Internamente la librería tiene una base de datos SQLite (pushdb) con dos tablas para mensajes salientes y mensajes entrantes que sea muy grandes (superiores a 4000Bytes). En la tabla de mensajes salientes *messages (*Figura 18) se encolan los mensajes que serán enviados al servidor con un identificador único **lid** que será enviado al servidor junto con el mensaje **content**, el campo **status** es un entero que indica si el mensaje ha sido recibido por el servidor o no(valor actual 0) cuando el servidor conteste con un ACK la recepción del mensaje el mismo será eliminado de la tabla *messages.* Como existen dispositivos con capacidades y características diferentes se creó la tabla de mensajes entrantes *receivedmessages*(Figura 19)

está destinada para los mensajes que lleguen y que sean muy grandes (superior a 4000Bytes), ya que estos mensajes serán enviados a la aplicación final a través de un *Intent* que tiene una capacidad limitada en el manejo de datos según el dispositivo. El Mensaje será almacenado en la tabla y sólo se enviará el id del registro correspondiente en BD para que se pueda obtener la información del lado del cliente. La tabla tiene como campos un **id** que será el identificador único e incremental, el campo **content** contiene el mensaje que llegó vía push y el campo **date** contiene la fecha y hora (en milisegundos) cando llegó el mensaje. Esta última tabla es de uso temporal y sólo sirve de puente para que el mensaje llegue a la aplicación cliente Final. En la Figura 18 se muestra la tabla *message* para mensajes salientes de la librería push en Android. En la Figura 19 la Tabla *receivedmessages* para los mensajes entrantes superiores a 4000Bytes.

Name	Type
id	INTEGER
lid	TEXT
content	TEXT
status	INTEGER

Figura 18 Tabla *message*.

Name	Type
id	INTEGER
content	TEXT
date	INTEGER

Figura 19 Tabla *receivedmessages*.

Los mensajes que lleguen vía push a través de la librería serán enviados vía Broadcast por lo que siempre se debe crear un *BroadcastReceiver* en la aplicación cliente para que capture y procese el mensaje recibido.

Recepción de mensajes

Los mensajes serán enviados a la aplicación final vía *Broadcast* utilizando el objeto *PushReceived* cuya representación se observa en el Código 6. Sus Atributos y métodos se definen en la Tabla 13 y 14 respectivamente. Este objeto contendrá la información recibida así

como otras variables que podrían resultar útiles al usuario. Esta es una clase que implementa la interfaz parcelable para que pueda ser pasada vía *Intent* al usuario final. Este objeto irá como extra en el *Intent* utilizando la clave *received* (se obtiene llamando al método getParcelable("received") en el BroadcastReceiver cómo se muestra en la Figura 20).

```
public class PushReceived implements Parcelable{
    private byte type;
    private String content;
    private int messageId;
    private byte headerByte;
    private boolean error;
    private boolean inDB;
    private long dbId;
    private long datetime;
    ...
}
```

Código 6 Objeto PushReceived que será capturado por el cliente final.

Atributos objeto PushReceived:

Atributo	Descripción
`private byte type;`	Variable que indica si el mensaje recibido es un mensaje que va destinado al la aplicación cliente 1, o 0 si el mensaje es un mensaje de control interno de la librería (mensaje de error, ack,conexión, etc).
`private String content;`	Variable que contiene el mensaje recibido desde el servidor. O mensaje de error según sea el caso.
`private int messageId;`	Id único del mensaje proveniente del servidor.
`private byte headerByte;`	Retorna el primer byte del mensaje. Útil para identificar el tipo de mensaje recibido.
`private boolean error;`	Indica si el mensaje recibido es un mensaje de error de la librería.
`private boolean inDB;`	Variable de uso interno que indica si el mensaje es superior a 4000Bytes y se encuentra en Base de datos(true), o si se viene en el campo content(false).

`private long dbId;`	En caso de que el mensaje sea muy grande y se encuentre en BD este campo contendrá el id asociado en registro en la tabla.
`private long datetime;`	Fecha y hora en milisegundos que indica cuando llegó el mensaje.

Tabla 13 Atributos de la Clase PushReceived

Métodos objeto PushReceived:

Método	Descripción
`private RR getXX;`	Todos los campos poseen sus respectivas funciones getXX para obtener el valor del campo deseado, en donde RR es el tipo de dato que retornará la función invocada.
`public String getContent(Context context);`	Función que retorna el contenido del mensaje. Esta función recibe como parámetro el contexto actual de la aplicación. La función se encargará de retornar el String que se encuentra en el campo content. En caso de que el mensaje sea grande y se encuentre en BD la función se encargará de ubicar el mensaje en la tabla *receivedmessages* y retornará el contenido del mismo al usuario final.

Tabla 14 Métodos de la clase PushReceived

Identificación de mensajes recibidos

La librería push envía la mayoría de los mensajes del servidor vía *Broadcast* al usuario final. Utilizando la función `getType()` se puede conocer si el mensaje recibido es un Mensaje enviado al servidor a la aplicación final (0 si es mensaje de control, 1 si es un mensaje destinado al usuario final). Con la función `getHeaderByte();` se puede saber qué tipo de mensaje ha sido el que se ha recibido y así poder realizar cualquier acción deseada (ejemplo enviar el nombre del dispositivo al servidor cuando se establezca la comunicación (`getHeaderByte()==_startByte`).

```
public class Push {
    public static final byte _headermessageByte=1;
    public static final byte _messageByte=2;
    public static final byte _messageACKByte = 6;
    public static final byte _startByte=15;
    public static final byte _beatByte=17;
    public static final byte _idByte=18;
    public static final byte _endByte=23;
    public static final byte _messageStartByte = 77;
    public static final byte _ErrorByte=4;

    public static final byte TypeControl=0;
    public static final byte TypeMessage=1;
}
```

Código 7 Clase Push con constantes de control

```
ublic class PushReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {

        PushReceived received=intent.getExtras().getParcelable("received");
        Log.d(G.TAG, "PushReceiver.On Received "+received.toString());
        Logger.AgregarEntrada(G.TAG,context, "PushReceiver.On Received "+received.toString());
```

Figura 20 Obtención del objeto PushReceived desde el BroadcastReceiver

En la Figura 21 se tiene una imagen con un fragmento del log de operaciones para mantener un tracking de lo sucedió. Desde el incio de conexión hasta el envío de mensajes pendientes con la recepción de sus respectivos ACK.

Línea 1: se inicia la conexión al servidor push.

Línea 3: una vez que se ha realizado la conexión la librería recibe el byte de inicio `_startByte` (15;***SI*** en la imagen). Al recibir esto la librería procede a enviar el Id del dispositivo.

Línea 5:la librería envía el Id del dispositivo al servidor utilizando el byte `_idByte` (18;**DC2** en la imagen).

Línea 6,7,9: se revisa por mensajes pendientes a ser enviados y lo envía utilizando el siguiente formato HHIIIIIIIIIIIISMMMMMM en dónde:

HH es el byte `_headermessageByte`(1;**SQH** en la imagen).

IIIIIIIIIII es el idLocal del mensaje almacenado en Base de datos (1429634723278 en la imagen)

S es el `_messageByte`(2;**STX** en la imagen).

MMMMMM es el mensaje enviado al servidor (M;Z2dn en la imagen).

Línea 10: se recibe un ACK desde el servidor con el byte `_messageACKByte`(6;**ACK** en la imagen). Indicando que el mensaje con id 1429634723278 fue recibido satisfactoriamente por el servidor (por lo tanto se elimina de la Base de datos local).

Línea 13, 14,15,16: en este punto está llegando el primer mensaje (StartByte) al receiver de la aplicación y la aplicación envía un mensaje que consiste en el nombre del dispositivo utilizando el `_messageByte`(2;**STX** en la imagen). En la línea 13,14 se puede ver el contenido del objeto PushReceived que recibirá el usuario final, en donde el campo type es 0 indicando que el mensaje que llegó es un mensaje de control (Mensaje de inicio de conexión).

Línea 17,18,19: llega al Receiver del usuario el ACK correspondiente al mensaje enviado en la línea 9. Con el campo type en 0 indicando que no es un mensaje de usuario(Mensaje de control ACK).

Línea 20-27: Proceso de envío de mensaje y recepción de ACK para mensaje con ID 1429634731377.

En la Figura 22 se extraen fragmentos del log con casos de recepción de menajes desde el servidor en este ejemplo se muestra el arribo de un mensaje superior a 4000Bytes (se almacena en BD) y el un mensaje meno a 4000Bytes llega en el objeto PushReceived .

Figura 21 Log con Recepción de mensajes a través del PushReceived.

Mensaje superior a 4000Bytes:

Línea 2: Arriba el mensaje al dispositivo con Id 4416 `_messageStartByte`(77;**M** en la imagen). El formato para el arribo de mensaje es M;IIII;RRRRRR en donde:

M es el byte que indica que es un mensaje dirigido al usuario.

IIII es el Id del mensaje en el servidor push

RRRRRR es el mensaje destinado al usuario final (Aplicación)

Línea 3: la librería envía un mensaje de confirmación al servidor _confirmationByte byte(67;**C** en la imagen). Con el Id del mensaje para indicar que fue recibido satisfactoriamente.

Línea 4: indica el tamaño del mensaje en superior a 4000Bytes.

Línea 6-9: el mensaje llega al Receiver de la aplicación y se observa como el campo *type* es igual a 1 (quiere decir que tipo mensaje), el campo *content* contiene el texto db y el campo dbId contiene el Id del registro que contiene el mensaje final en BD.

Mensaje menor a 4000Byes:

Línea 11: Arriba el mensaje al dispositivo con Id 4415 `_messageStartByte`(77;**M** en la imagen). El formato para el arribo de mensaje es M;IIII;RRRRRR en donde:

M es el byte que indica que es un mensaje dirigido al usuario.

IIII es el Id del mensaje en el servidor push

RRRRRR es el mensaje destinado al usuario final (Aplicación)

Línea 12: la librería envía un mensaje de confirmación al servidor _confirmationByte byte(67;**C** en la imagen). Con el Id del mensaje para indicar que fue recibido satisfactoriamente.

Línea 13: indica el tamaño del mensaje en menor a 4000Bytes.

Línea 14-18: el mensaje llega al Receiver de la aplicación y se observa como el campo *type* es igual a 1 (quiere decir que tipo mensaje), el campo *content* contiene el mensaje original y el campo dbId es 0 (no es almacenado en Base de datos).

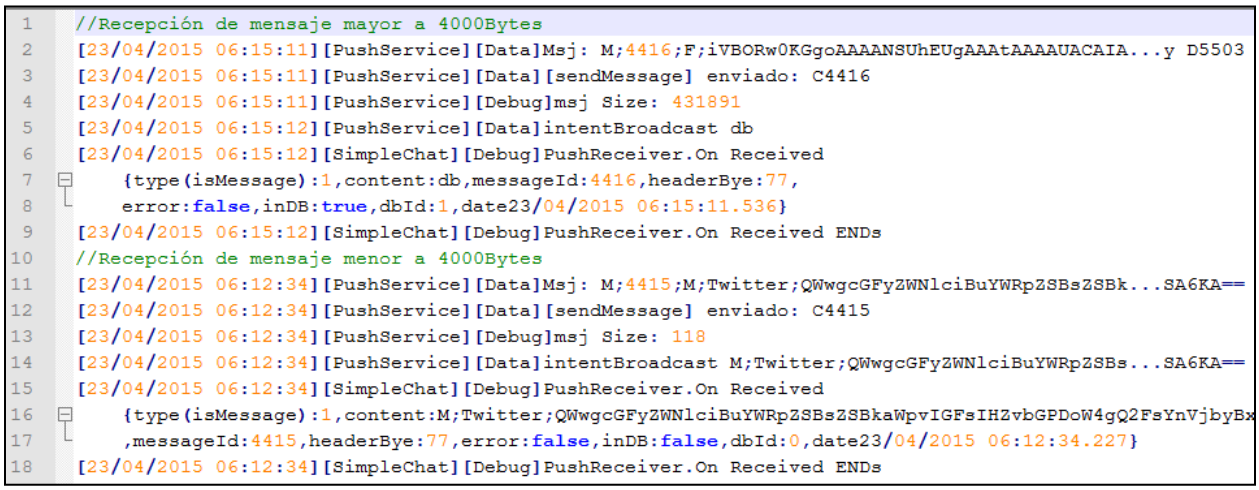


Figura 22 Recepción de mensajes desde el servidor

Detección de errores

La librería es capaz de pasar los errores de conexión al usuario para que este haga lo que crea conveniente en caso de que no haya comunicación con el servidor. La función `isError()` indicará si el mensaje recibido es un mensaje de error o no. Si el mensaje es un error la función `getContent()` retornará un String con la descripción del error (Actualmente, solo se envían mensajes de error de conexión). En la Figura 23 se muestra el Log con un ejemplo del Objeto PushReceived en donde el campo error es igual a true y el campo content obtiene en texto el mensaje de error.

Si durante el envío de mensajes hacia el servidor se cae la conexión o esta no está disponible los mensajes son almacenados en la Base de datos local y se envían al servidor cuando se vuelva a establecer la conexión.

Por defecto la librería está configurada para que en caso de una falla en a conexión con el servidor se realicen intentos de conexión con tiempos de espera que van incrementando de acuerdo a la cantidad de reintentos (Exponencial BackOff), el tiempo de espera irá incrementando en base a la formula `((long) Math.pow(2, retryCount) * 100L)` donde retryCount es la cantidad de intentos que se tiene tratando de realizar la reconexión. Este tiempo de espera es en milisegundos y el valor máximo configurado entre cada reintento es `18000000` ms que equivalen a 30 min.

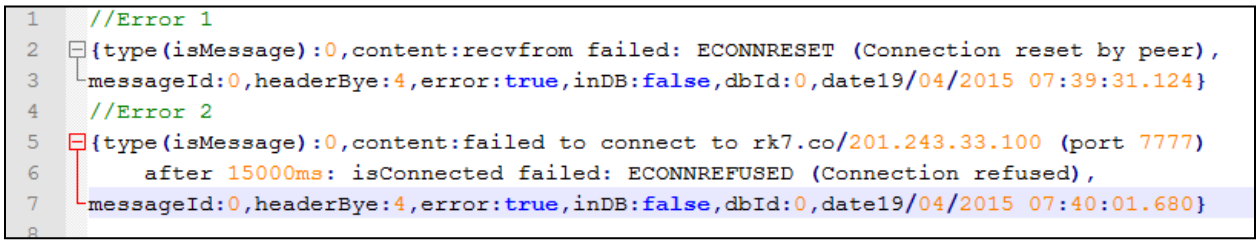


Figura 23 objeto PushReceived con mensaje y bit de error

Inicio y Configuración

Para utilizar la librería en Android se debe incluir el archivo .jar o referenciar el proyecto como librería a la nueva aplicación Android que se va a desarrollar. Lo primero que se debe hacer es inicializar los parámetros de conexión hacia el servidor. En la Figura 24 se observa un ejemplo en la inicialización en los parámetros de la librería e inicio del servicio. En la línea 234 se inicializan los parámetros de conexión y funcionamiento utilizando e método estático SetInit(…) de la librería. El primer parámetro es el contexto actual de la apliación , el segundo parámetro es el nombre del host o dirección Ip en donde estará corriendo el servicio , el tercer parámetro es el puerto por donde estará escuchando el servicio, el cuarto parámetro es un Id único que representa a la aplicación y el quinto parámetro indica si se debe mantener el procesador activo durante la ejecución del servicio. Una vez que se haya inicializado la librería se puede proceder a ejecutar el servicio que mantendrá la conexión con el servidor. El servicio está configurado con los valores del intent filter, timeout, y ejecución de la espera incremental por defecto pero estos valores son configurables y se pueden enviar como extras utilizando un Bundle para el servicio. Los identificadores a utilizar, los tipos de datos y los valores por defecto pueden.

```
227  @Override
228  protected void onCreate(Bundle savedInstanceState) {
229      super.onCreate(savedInstanceState);
230      setContentView(R.layout.activity_main);
231
232      ...
233
234      ron.tesis.PushService.ConnectionService.SetInit(getApplicationContext(),"rk7.co",7777,1,false);
235      Intent serviceIntent =new Intent(this,ron.tesis.PushService.ConnectionService.class);
236      Bundle extras = new Bundle();
237      extras.putString("INTENT_FILTER", "ron.tesis.Ejemplo.ChatMSJ");
238      extras.putInt("TIMEOUT",15);
239      extras.putBoolean("DISABLEBACKOFF", true);
240      serviceIntent.putExtras(extras);
241      startService(serviceIntent);
242      ...
```

Figura 24 Inicialización de la librería Push

Permisología y configuración en el AndroidManifest

En la figura 25 se muestran las permisologías necesarias para el correcto funcionamiento de la librería. En la tabla 16 se detalla el uso de cada una de ellas. La versión mínima de la aplicación debe ser la 9 y se debe especificar en el tag *android:minSdkVersion="9"* . En La Figura 26 se indica cómo debe ser registrado el servicio (línea 47) que ejecuta la librería y los Receiver que utiliza la aplicación. Se debe referenciar el Receiver *PushService.ConnectionReceiver* que es el encargado de registrar los cambios de estado en la conectividad del teléfono y de iniciar o detener el servicio si la existe o no una conexión respectivamente. En las líneas 42-46 se encuentra registrado un Receiver Local que es el encargado de recibir los mensajes que la librería push envíe a la aplicación. El valor del intent filter `ron.tesis.PushService.MSJ` es el que viene configurado en la librería por defecto, si se cambia este parámetro también debe cambiarse al momento de iniciar configurando el parámetro *INTENT_FILTER* en el bundle (ver tabla 15).

Nombre	Etiqueta	Tipo de Dato	Valor por defecto	Descripción
Timeout	"TIMEOUT"	Entero	15	Tiempo de espera máximo para la conexión hacia el servidor

Nombre	Etiqueta	Tipo de Dato	Valor por defecto	Descripción
TimerCheck	"TIMERCHECK"	Entero	30	Tiempo para enviar los paquetes de conexón para verificar si la conexión sigue activa.
Intent filter	“INTENT_FILTER”	String	"ron.tesis.PushService.MSJ"	Intent filter utilizado por el BroadcastReceiver para la recepción de los mensajes que envía la librería
Espera incremental	"DISABLEBACKOFF"	Boolean	false	Si es verdadero, los intentos de reconexión al servidor no serán realizados de forma incremental. Los intentos de reconexión serán constantes y el tiempo entre cada intento serán 30 Segundos hasta que se logre establecer la conexión.

Tabla 15 Valores configurables que pueden ser pasados al servicio a través de un Bundle

```
<receiver android:name="ron.tesis.simplechat.receiver.PushReceiver">
        <intent-filter>
            <action android:name="ron.tesis.PushService.MSJ"/>
        </intent-filter>
</receiver>
```

Código 8 Registro del *reciver* en el AndroidManifest (XML)

```
10
11      <!-- Permisos Lib push -->
12      <uses-permission android:name="android.permission.INTERNET" />
13      <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
14      <uses-permission android:name="android.permission.WAKE_LOCK" />
15      <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
16      <uses-permission android:name="android.permission.READ_PHONE_STATE" /> <!-- Onbtener id telefono -->
17      <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"></uses-permission>
18
```

Figura 25 Permisologías necesarias para el uso de la librería Android.

Permiso	Uso
INTERNET	Por tratarse de una librería de comunicación se requieren permisos para el uso de las interfaces de red del equipo.
ACCESS_NETWORK_STATE ***ACCESS_WIFI_STATE***	Requerido para conocer el estado de la red actual, saber si está conectado o desconectado y activar/desactivar el servicio según sea el caso.
WAKE_LOCK	Requerido para mantener el cpu activo mientras el dispositivo está en modo de reposo.
WRITE_EXTERNAL_STORAGE	Necesario para escribir en el log de pruebas.
READ_PHONE_STATE	Requerido para obtener información del teléfono, como el IMEI y la MAC (utilizado para generar el identificador único).

Tabla 16 Uso de permisologías para la librería Push en Android

```
34
35          <!-- Servicios Lib push -->
36          <receiver android:name="ron.tesis.PushService.ConnectionReceiver">
37           <intent-filter>
38                <action android:name="android.net.conn.CONNECTIVITY_CHANGE"/>
39                <!-- <action android:name="ron.tesis.PushService.RST"/> -->
40           </intent-filter>
41          </receiver>
42          <receiver android:name="ron.tesis.simplechat.receiver.PushReceiver">
43           <intent-filter>
44                <action android:name="ron.tesis.PushService.MSJ"/>
45           </intent-filter>
46          </receiver>
47          <service  android:enabled="true"  android:name="ron.tesis.PushService.ConnectionService" />
48      </application>
49
```

Figura 26 Registro del servicio y los BroadcastReceiver

3.8 Base de Datos (servidor)

Diseño de la Base de Datos

Los mensajes deben ser almacenados en BD, de este modo si el dispositivo no se encuentra conectado al momento de enviar el mensaje el mismo es almacenado y enviado cuando el dispositivo vuelva a conectarse con el servidor.

La Base de Datos constará inicialmente de 5 tablas (como se muestra en la Figura 27.) los nombres de las mismas estará precedida de la palabra **push_**

- **push_Device:** Se encargará de almacenar Todos los dispositivos que se conecten al servidor, cada dispositivo tendrá un identificador único **(Id)** que se obtiene al momento de ejecutar el servicio en el lado del cliente (librería java), la fecha de registro **(RegisterDate)** que contendrá la fecha y la hora en la que el dispositivo realizó la primera conexión, Ultima Conexión**(LastConnection)** para indicar la fecha en la que el dispositivo se conectó por última vez, Alias **(Alias)** Campo de texto utilizado para asignar un nombre amigable.

- **push_Mensssage:** Esta tabla se encargará de almacenar todos los mensajes generados y que posteriormente serán enviados a los dispositivos seleccionados. Cada Mensaje consta de un Identificador único Autogenerado **(Id)** , un campo de texto que contendrá el mensaje a enviar **(Message)** La fecha de creación del Mensaje **(Date)** y la fecha en que el mensaje expirará(Tiempo de vida del mensaje) **(DueTime),**un campo de tipo bit que indicará si el mensaje debe recibir una confirmación para verificar que llegó al destinatario(**Confirmation**), el campo (**IdDevice)** tendrá el Id del dispositivo origen que generó el mensaje y finalmente el campo (**LocalId**) es un identificador Interno que utilizará la librería Android para el control de los mensajes.

- **push_Message_Device:** Tabla que mantendrá la relación entre los mensajes enviados y los dispositivos a los que debe llegar el mismo, esta tabla contendrá como clave primaria el Id del mensaje y el Id del dispositivo al cual se le va a enviar el mensaje **(IdMessage)** y **(IdDevice)** adicionalmente existe el campo **(Delivered)** que indicará (en caso de que así

sea configurado en el mensaje, Campo Confirmación de la tabla push_Message sea igual a 1) si el dispositivo recibió el mensaje o no.

- **push_Group[Futuro]:** tabla destinada a la creación de Grupos para el envío de Mensajes push masivo. Esta tabla contendrá un Identificador único para el grupo **(Id)** Un campo de texto para el nombre del grupo (**Name)** y un campo de descripción **(Desc)** para la descripción del grupo

- **push_Device_Goup[Futuro]:** Tabla que contendrá la relación de los equipos que pertenecen a los grupos creados, esta tabla tiene como únicos campos y clave primaria el identificador del grupo y el identificador del equipo **(IdGroup)** y **(IdDevice).**

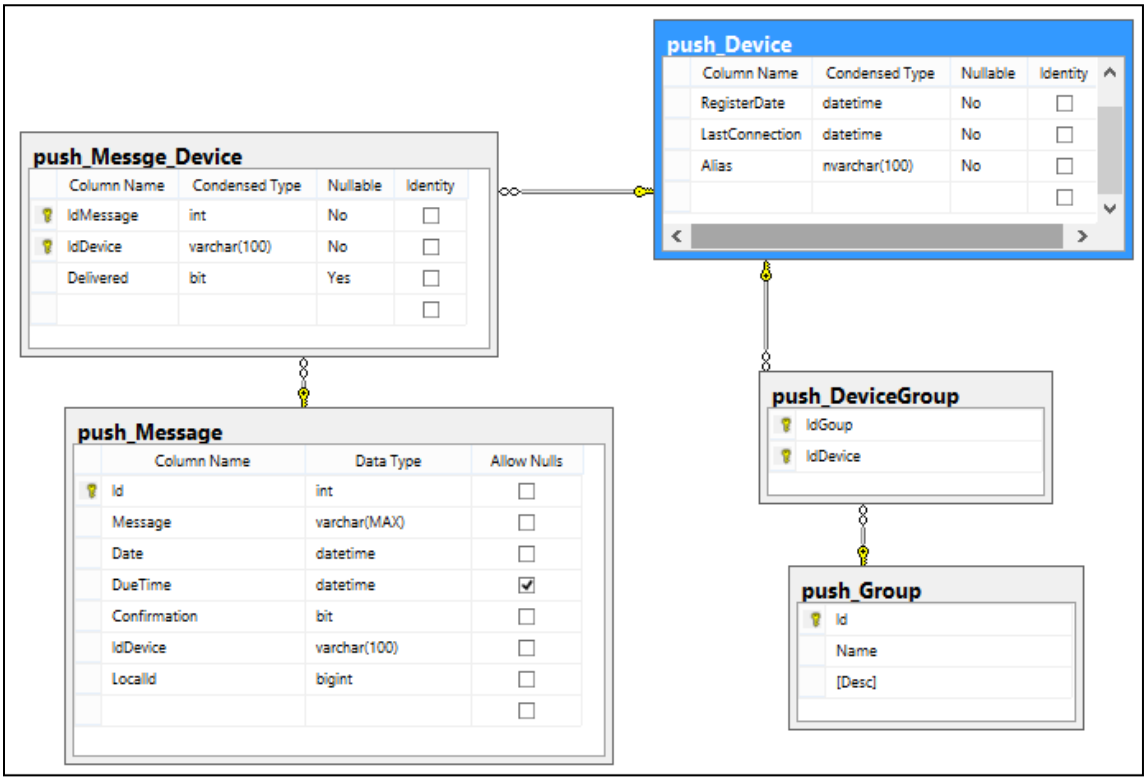


Figura 27 ER Tablas Push

Capítulo 4: Diseño y Desarrollo de casos de estudio utilizando la solución propuesta.

En este capítulo se describen tres aplicaciones que fueron desarrolladas para utilizar las librerías creadas en el capítulo 3 y demostrar su uso, comportamiento y hacer las respectivas pruebas en cuanto a funcionalidad y manejo de errores.

4.1 WebSMS

WebSMS es portal web que permitirá a los usuarios registrados realizar el envío de mensajes de texto a múltiples usuarios de teléfonos celulares a nivel nacional. El envío será realizado a través de un teléfono móvil Android que tendrá instalado una aplicación desarrollada en Java que utilice la librearía creada en el capítulo 3. Se deben crear los componentes necesarios utilizando las librearías diseñadas previamente para que se pueda mantener una conexión entre el dispositivos clientes, el servidor y la página web-servidor. Se debe crear un "Protocolo" de comunicación que para que puedan ser interpretados los datos que se envían desde el dispositivo cliente al servidor.

Las partes en este portal Web son:

- Portal Web
- Aplicación Servidor(Windows)
- Aplicación Cliente(Android)

4.1.1 Modelo de Caso de Uso

A continuación en la Figura 28 se muestra el diagrama de caso de uso.

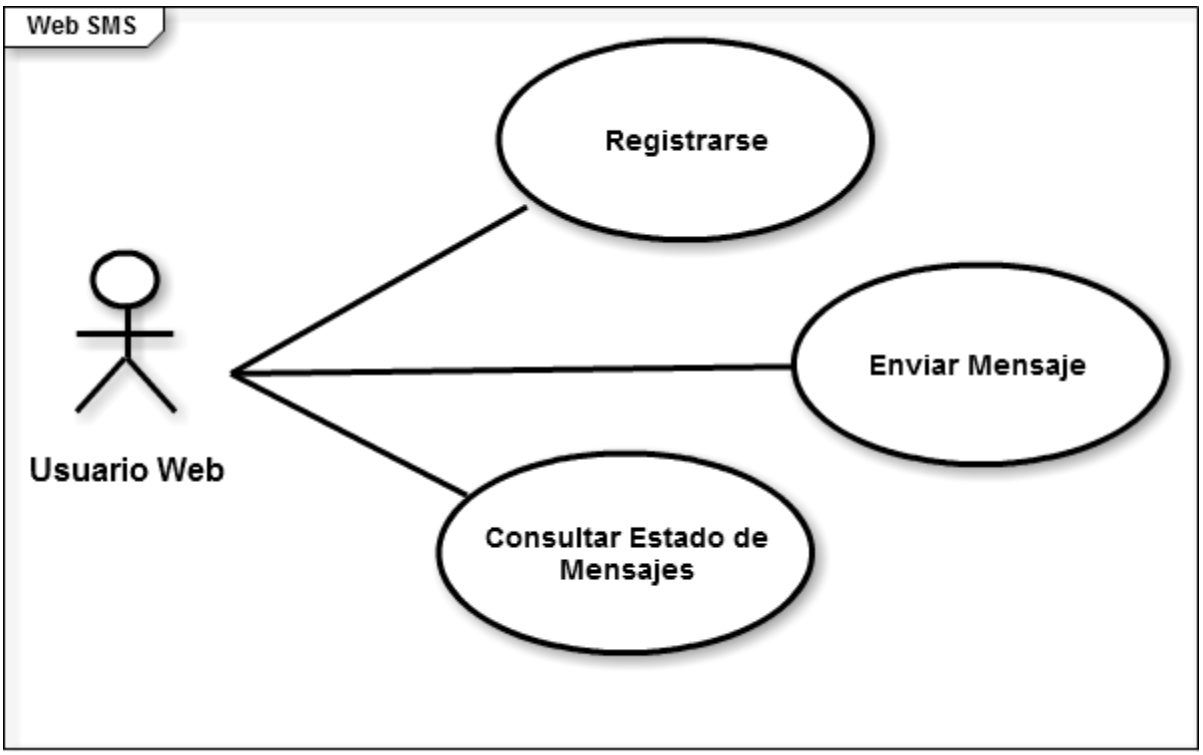


Figura 28 Caso de Uso usuario Web

- **Registrarse:** caso de uso mediante el cual el usuario se registra creando una cuenta en el portal web.
- **Enviar Mensaje:** caso de uso mediante el cual el usuario crea la petición para el envío de mensajes de texto.
- **Consultar Estado de Mensajes:** caso de uso mediante el cual el usuario consulta el estado actual de la petición creado para el envío de mensajes de texto.

4.1.2 Diagrama de Secuencia

El siguiente diagrama de la Figura 29, 29.1 se muestra el flujo completo entre la página web, el dispositivo cliente y el servidor.

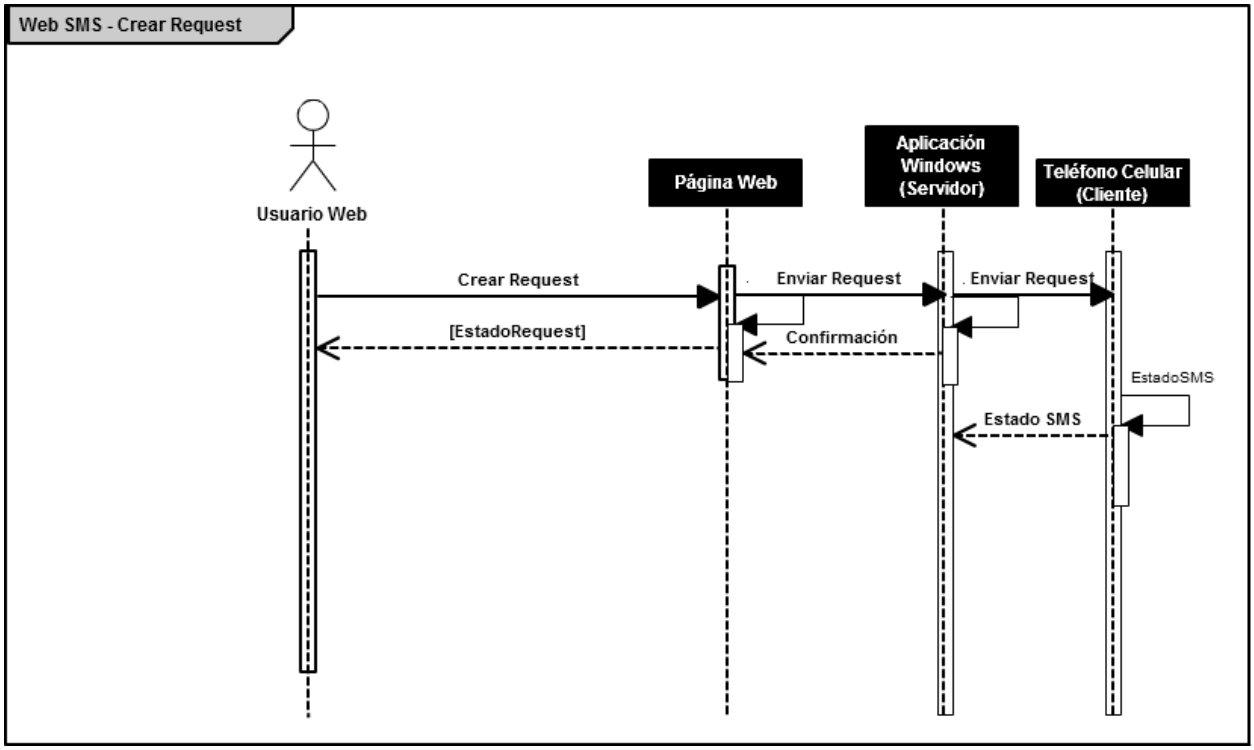


Figura 29 Diagrama de secuencia Web SMS

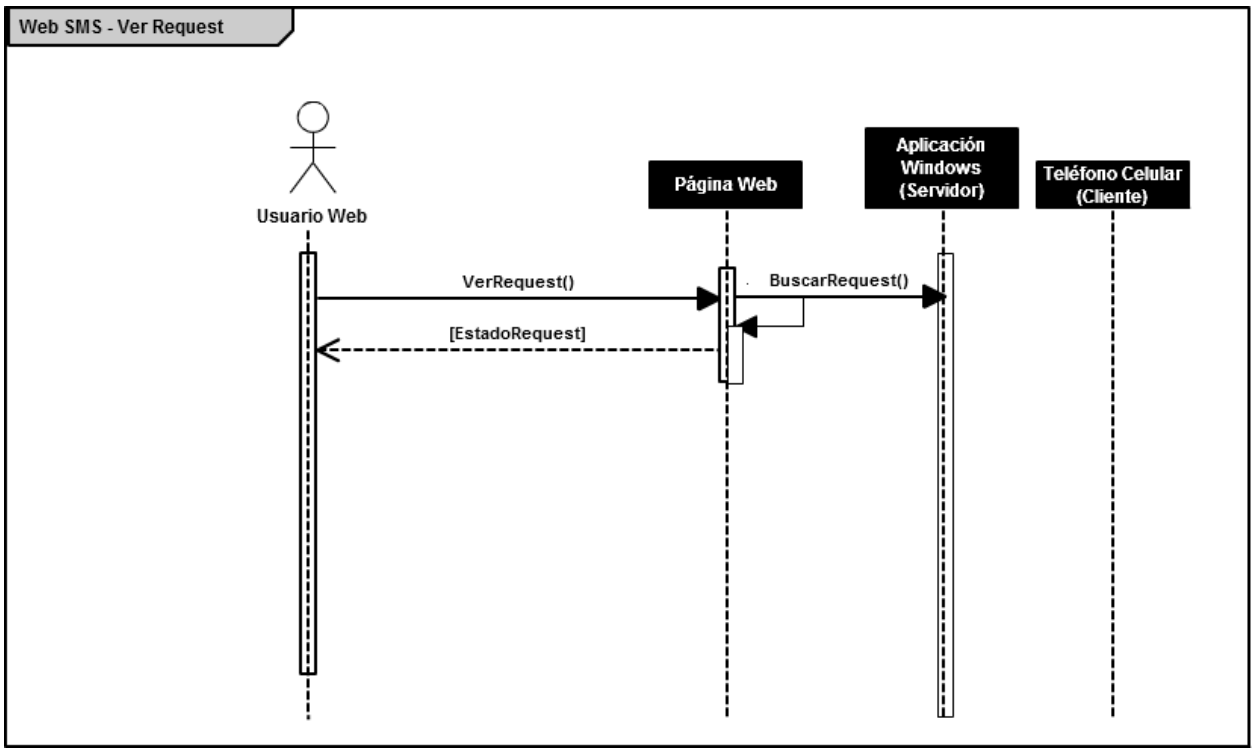


Figura 29.1 Diagrama de Secuencia Ver Request

4.1.3 Base de Datos

Adicionalmente a las tablas necesarias para el funcionamiento del servicio push que se especifican en el capitulo anterior y se importan utilizando un Script, se deben crear la estructura necesaria en Base de datos que permita la persistencia de datos para los usuarios registrados, Mensajes de texto enviados, estado de mensajes enviados para que todo esto pueda ser accedido y consultado desde la página web diseñada para tal fin. Estas tablas también serán accedidas por la aplicación Servidor que será la encargada de marcar el Estado final de los mensajes enviados. En la Figura 30 se muestra el Diagrama entidad Relación de las tablas.

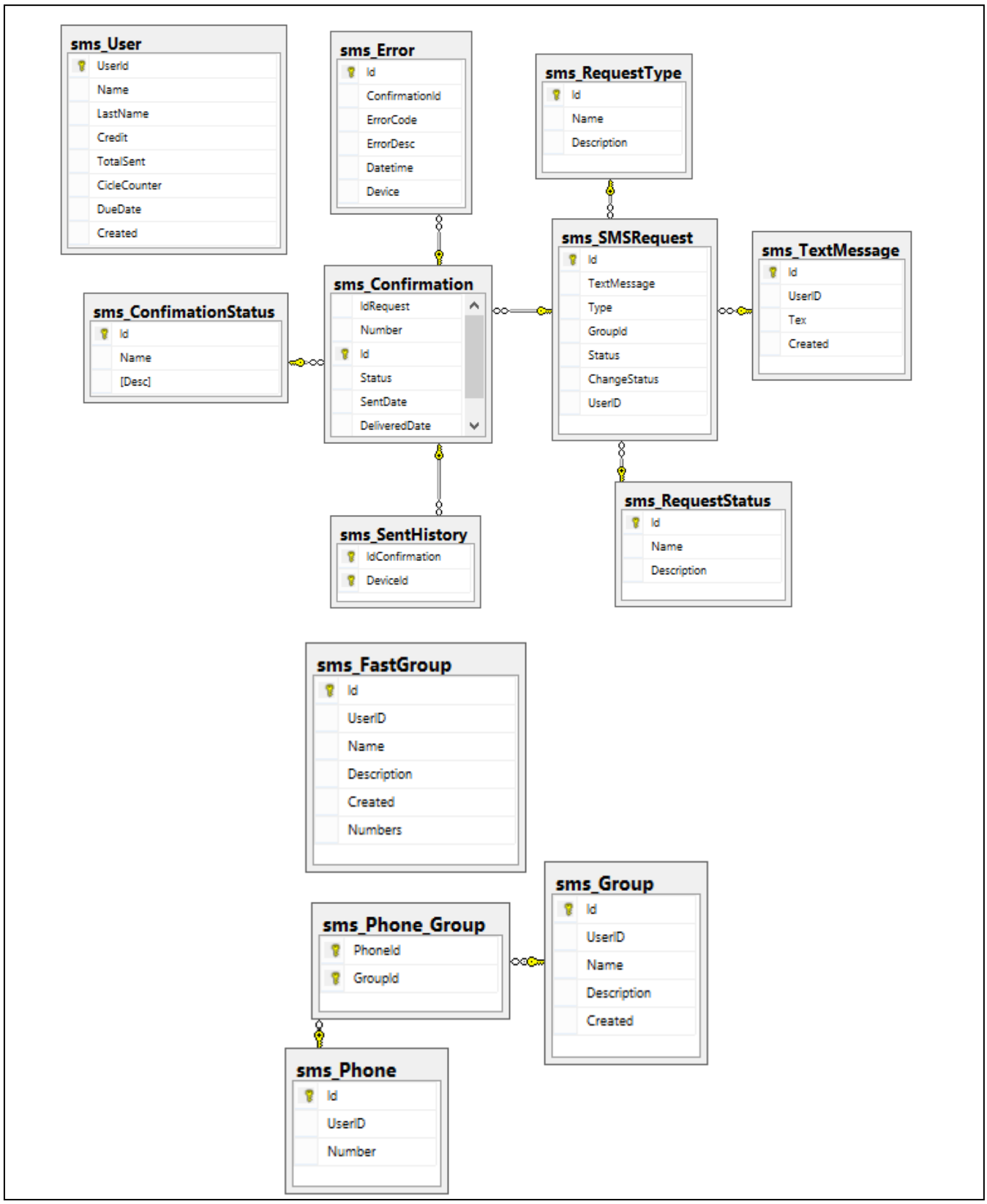


Figura 30 ER Diagrama WebSMS

Las tablas asociadas a la página web y servicio de mensaje de texto estarán precedidas de la palabra **sms_.**

- **sms_User:** Tabla que contendrá lo referente al perfil de usuario registrado desde la página web. Esta tabla posee como clave primaria la columna (**UserId**) que servirá de identificación única para cada usuario que se registre al sistema. Contiene datos básicos como el Nombre (**Name**) y Apellido (**LastName**). Cada Usuario en la página dispondrá de una cantidad de créditos o "cupos" para el envío de mensajes de texto. La cantidad de mensajes disponible para el envío será representado con la columna (**Credit**).Los campos (**TotalSent**) y (**CicleCounter**) indican la cantidad total de mensajes enviados desde que se registró el usuario y la cantidad de mensajes enviados al mes respectivamente. El campo (**DueDate**) y (**Created**) son de tipo Datetime y nos indica la fecha Límite para el ciclo de envío de mensaje de texto(fecha de corte mensual para el CicleCounter) y la fecha de creación del usuario respectivamente.

- **sms_TextMessage:** Tabla que almacenará el contenido de los mensajes de texto creados. Esta tabla tiene como clave primaria un entero (**Id**) que servirá de identificación única para cada mensaje creado. Este mensaje estará asociado al usuario Web que lo ha creado a través del campo (**UserId**), cuyo valor será el Id del usuario que creó el mensaje. El campo (**Text**) contendrá el contenido del mensaje de texto y el campo (**Created**) representa la fecha de creación de dicho mensaje en el sistema.

sms_RequestType: Tabla Maestra que contendrá los tipos de Requests que se puedan crear en la página. Tiene como clave primaria un id único (**Id**), un nombre (**Name**) y una descripción del tipo de request (**Description**). En la Tabla 18 se describen los tipo de request.

Id	Nombre	Descripción
1	FastGroup	Grupos rápidos previamente almacenados momento de crear el request para el envío de mensaje. Estos grupos pueden ser reutilizados y son almacenados en la tabla **sms_FastGroup.**
2	Group	Grupos previamente almacenados antes de crear el request para el envío de mensaje. Estos grupos pueden ser utilizados para el envio de mensaje de texto por lotes y son almacenados en la tabla **sms_Group.**
3	NewFastGroup	Grupos rápidos Nuevos y creados al momento de crear el request para el envío de mensaje. Estos grupos pueden ser reutilizados y son almacenados al momento de crear el request en la tabla **sms_FastGroup.**
4	SingleNumber	Tipo de request para el envío de mensaje de texto a un solo numero en específico.

Tabla 17 Tabla sms_RequestType

sms_RequestStatus: Tabla Maestra que contendrá el estatus de los request que sean creados. Tiene como clave primaria un id único (**Id**), un nombre (**Name**) y una descripción que para el estado del request (**Description**). En la Tabla 19 se describen los estados del request.

Id	Nombre	Descripción
1	Pendiente	Indica que el Request ha sido creado y que está pendiente a ser finalizado.
2	Procesando	Indica que el Request ha comenzado a ser procesado y los mensajes se están enviando.
3	Finalizado	Indica que el Request ha Finalizado exitosamente.
4	Advertencia	Indica que el Request ha finalizado pero que uno o más mensaje no pudo ser enviado exitosamente.

Tabla 18 Tabla sms_RequestStatus

- **sms_SMSRequest:** Tabla que contiene el request creado por el usuario a través de la página web. Esta tabla tiene como clave primaria el campo de tipo entero **(Id)** único que

servirá para identificar los requests. El campo **(TextMessage)** es un entero y clave foránea de la tabla sms_TextMessage que asocia el request con el mensaje de texto creado. El campo **(Type)** es una entero y clave foránea de la tabla sms_RequestType e indicará que tipo de request es el que se ha creado. El campo **(GroupId)** es un entero que indicará el Id del grupo **(sms_Group)** o grupo rápido (sms_FastGroup). **(Status)** Es un entero y clave foránea de la tabla sms_RequestStatus e indicará el estado del Request actual. **(ChangeStatus)** es un campo de tipo Datetime e indicará la fecha la fecha y hora en la que se actualizó el estado del request actual. **(UserID)** es el Id del usuario que creó el request.

- **sms_ConfirmationStatus:** Tabla Maestra que contendrá el estatus de los mensajes de texto individuales que sean creados y enviados. Tiene como clave primaria un id único (**Id**), un nombre (**Name**) y una descripción que para el estado del request (**Desc**).En la Tabla 20 se definen los campos de la Tabla

Id	Nombre	Descripción
0	Waiting	Indica que el mensaje está en cola y aún no ha sido enviado.
1	Sent	Indica que el mensaje ha sido enviado.
2	Delivered	Indica que el mensaje ha sido enviado y ha sido recibido por el teléfono destino
3	Error	Indica que ocurrió un error al intentar enviar el mensaje
4	Locked	Indica que el Mensaje se ha enviado más de 3 veces y ha sido bloqueado para evitar futuros intentos automáticos.

Tabla 19 Tabla sms_ConfirmationStatus

- **sms_Confirmation:** Tabla que contendrá la información detallada del estatus de cada mensaje de texto que se haya enviado. Posee como Identificador único el campo numérico (**Id**). El campo (**IdRequest**) es una clave foránea a la tabla sms_SMSRequest y asocia el mensaje individual con el Request asociado. (**Number**) es un String que contiene el número telefónico a donde será enviado el mensaje de texto. (**Status**) Entero

y clave foránea a la tabla *sms_ConfirmationStatus* e indica el estado actual del mensaje. (**SentDate**) indica la fecha y hora de envío del mensaje. (**DeliveredDate**) Indica la fecha y hora en la que el mensaje ha sido recibido.

- **sms_SentHistory:** Tabla que contendrá un histórico de los mensajes enviados y el dispositivo asociado. Tiene una clave primaria compuesta por el Id de confirmación (**IdConfirmation**) y el String (**DeviceId**) que indica el Id del dispositivo final que envió el mensaje.
- **sms_Error:** Tabla que almacenará un histórico de los errores en el envío de mensaje que pueda ocurrir. Tiene como identificador único el campo numérico (**Id**). (**ConfimationId**) es clave foránea a la tabla *sms_Confirmation* e identifica el mensaje enviado. (**ErrorCode**) y (**ErrorDesc**) indican el código de error y su descripción respectivamente. El campo (**Datetime**) indica la hora y fecha en la que se presentó el error y el campo (**Device**) contiene el identificador del dispositivo que falló en enviar el mensaje.
- **sms_FastGroup**: Tabla que contendrá los grupos rápidos que han sido creados al momento de hacer el request. Estos grupos se almacenan y pueden ser reutilizados. La tabla tiene como clave primaria el campo (**Id**). El campo (**UserId**) está asociado al usuario que creo el grupo. (**Name**) es un nombre o alias que se le da al grupo.(**Description**) contiene una breve descripción del grupo.(**Created**) indica la fecha de creación y (**Numbers**) contiene los números asociados al grupo. Estos números son separados por ‘,’ o ‘;’.
- **sms_Group:** Tabla que contendrá los grupos que han sido creados antes de crear el request para que puedan ser reutilizados. La tabla tiene como clave primaria el campo (**Id**). El campo (**UserId**) está asociado al usuario que creó el grupo. (**Name**) es un nombre o alias que se le da al grupo.(**Description**) contiene una breve descripción del grupo.(**Created**) indica la fecha de creación.

- **sms_Phone:** Tabla que almacenará los números telefónicos asociados a un grupo. Tiene como clave primaria el campo numérico (**Id**). El campo (**UserId**) es id asociado al usuario que creó el grupo y (**Number**) es el número telefónico que será agregado al grupo.
- **Sms_Phone_Group:** Tabla de relación que asocia el número telefónico con el grupo creado. Tiene una clave primaria compuesta (**PhoneID**) y (**GroupId**) cuyas claves foráneas son el Id del número telefónico en la tabla *sms_Phone* y el id del Grupo creado en la tabla *sms_Group*.

4.1.4 Portal Web

Página Web asp .NET que procesará el registro de usuario y proporcionará la interfaz web para la carga, envío y visualización de estatus de los mensajes de texto que hayan sido enviados. La página web se comunicará con el servicio push a través del puerto de comandos que estará escuchando peticiones entrantes.

4.1.4.1 Interfaz del portal Web

En esta sección se muestra la interfaz de la página Web SMS

Página principal (Home)

Pantalla principal del sistema Web SMS que muestra los enlaces para iniciar sesión o crear una cuenta nueva. En la Figura 31 se muestra la pantalla inicial.

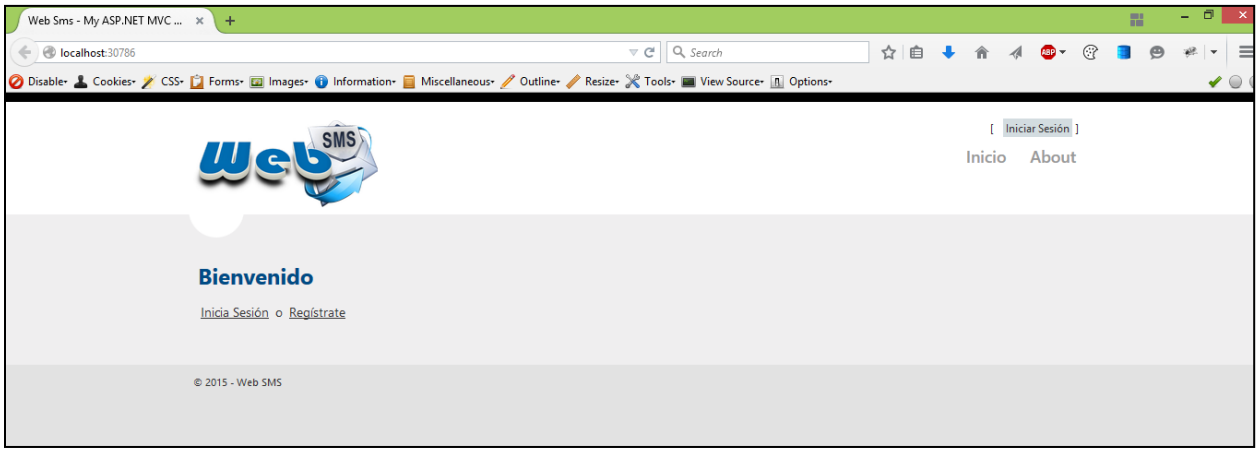


Figura 31 – Página Principal Web SMS

Registro

Pantalla que muestra el formulario de registro para el sistema WebSMS. El usuario debe rellenar el formulario y presionar el botón crear para crear una cuenta nueva. En la Figura 32 se muestra la pantalla de registro.

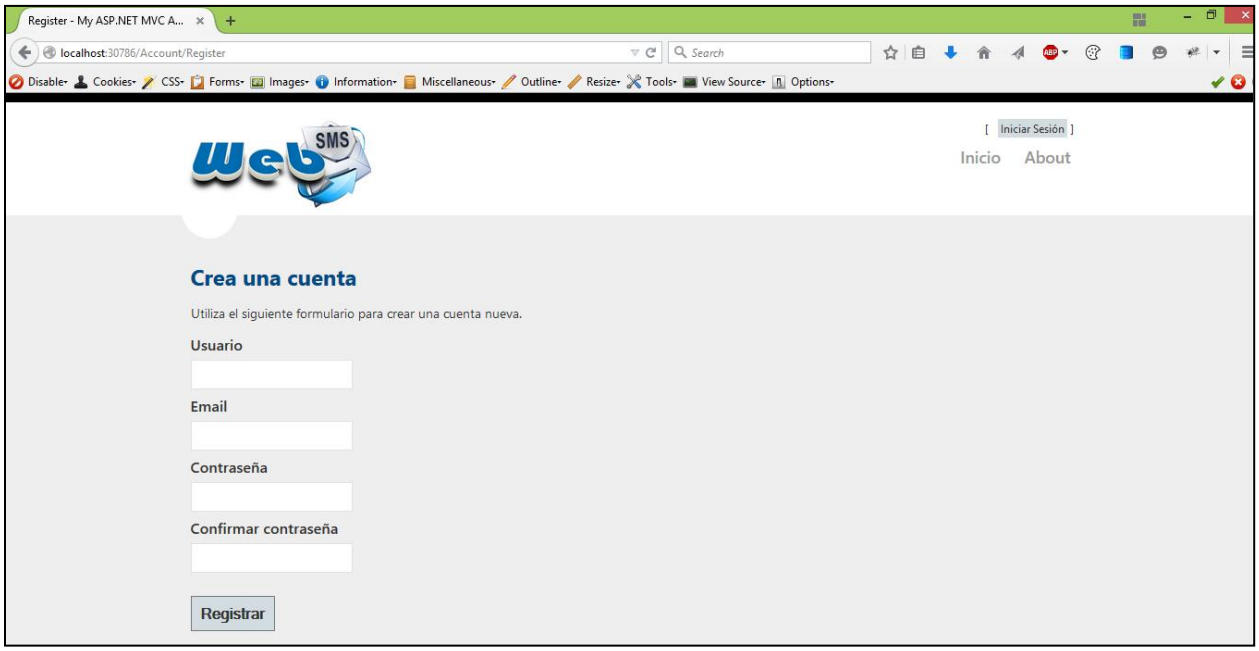


Figura 32 – Página de Registro WebSMS

Inicio Sesión

Pantalla que muestra el formulario de inicio de sesión del sistema Web SMS. El usuario debe ingresar los datos requeridos para iniciar sesión en el sistema. En la Figura 33 se muestra la página de inicio de sesión.

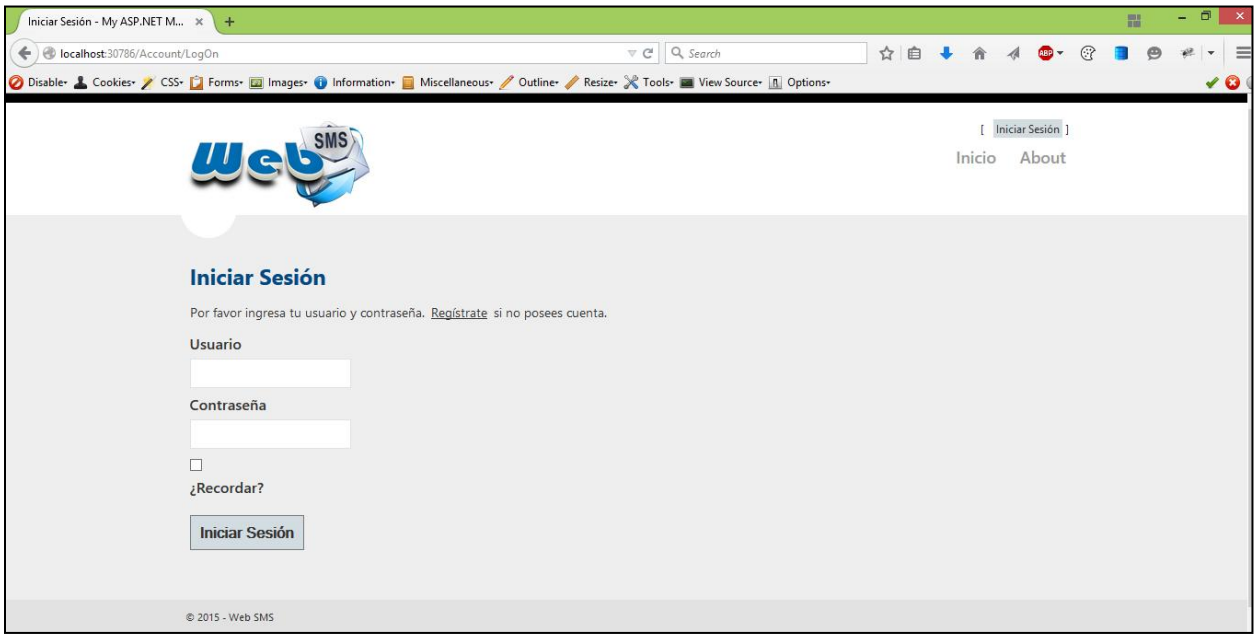


Figura 33 – Página de Inicio de sesión Web SMS

Pantalla de Usuario Principal

Pantalla que muestra las opciones del usuario principal. En la Figura 34 se muestra la pantalla de usuario principal.

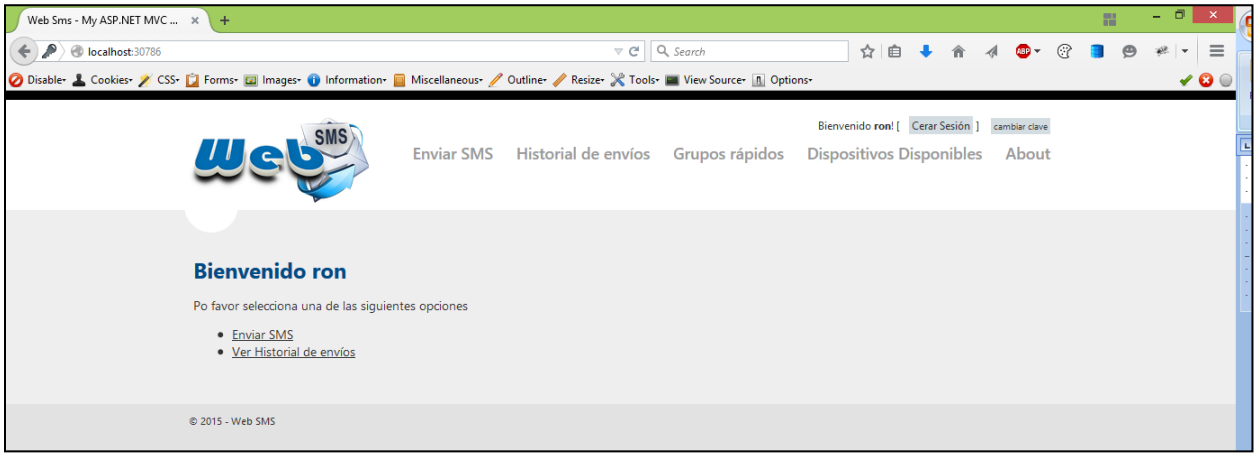


Figura 34 – Página del usuario principal.

Pantalla de envío de SMS

Pantalla para el envío de SMS. En la Figura 35 se muestra la pantalla consta de un campo de texto para introducir el texto del mensaje y 3 opciones para la selección de los destinatarios:

FastGroup: Grupos rápidos recientes.

Group: Grupos creados previamente.

Nuevo: se ingresa una lista de números para crear un nuevo grupo rápido.

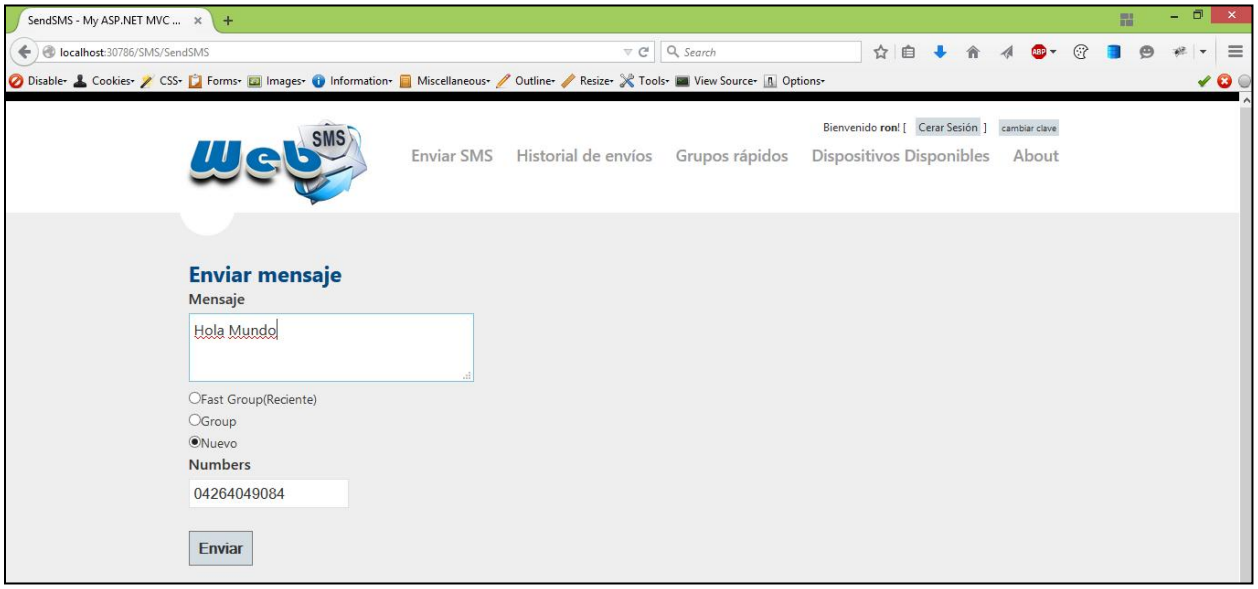


Figura 35 Página la creación del request y envío del mensaje.

Pantalla la consulta de estado del request

Pantalla para consultar el estado del request. Ver los mensajes enviados, los que fueron entregados, los que presentaron errores, ver la hora de envío y en caso de ser posible la hora en entrega. En la Figura 36 se muestra la pantalla de consulta de request.

4.1.5 Aplicación Servidor

Aplicación o servicio en Windows que se encargará de recibir las peticiones que llegan desde la página web, seleccionar el dispositivo móvil que va a enviar el mensaje y pasar los datos necesarios a este dispositivo (vía push) para que se encargue del envío de mensaje. Esta aplicación debe monitorear los mensajes que lleguen desde el dispositivo para actualizar en BD el estado de los mensajes (Enviados, Recibidos, Error).

4.1.5.1 Modelo de Caso de Uso

A continuación en la Figura 37 se muestra el diagrama de caso de uso para el Servidor WebSMS.

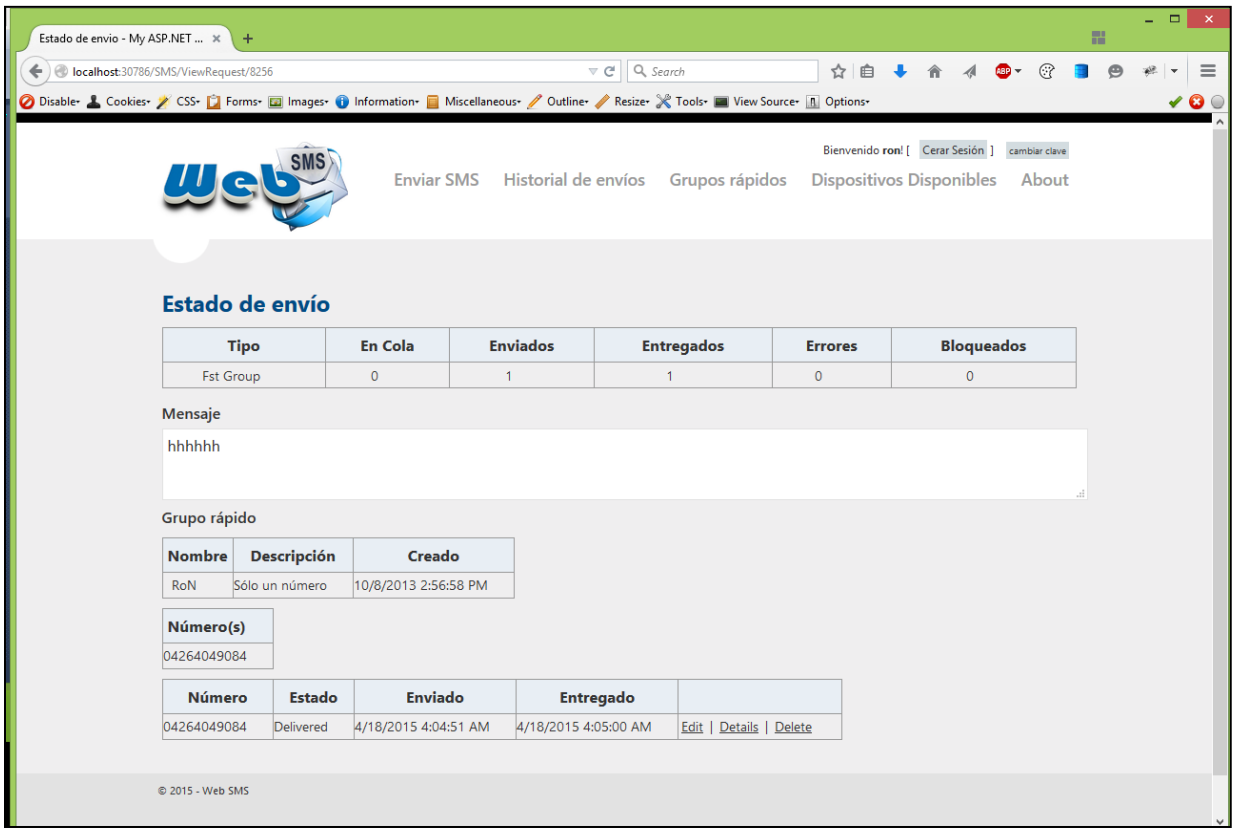


Figura 36 – Pagina para la consulta del request realizado

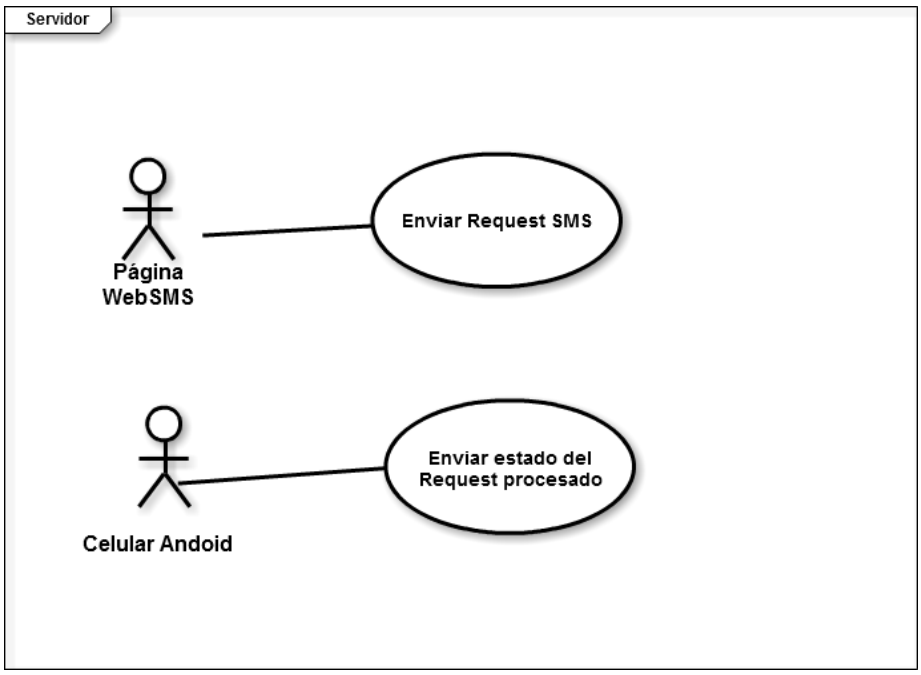


Figura 37 Modelo de caso de uso del servidor push WebSMS

Enviar Request(WebSMS): caso de uso realizado por la página web en donde esta envía al servidor el request del usuario a través del puerto de comandos.

Enviar estado del Request procesado: caso de uso ejecutado por el cliente Android en el cual envía el estado del mensaje de vuelta al servidor para que este pueda actualizar estos valores en la Base de datos sobre las tablas correspondiente al request creado por el usuario.

4.1.5.2 Implementación del servidor

En la Figura 38 se utiliza la librería desarrollada (push.dll) y se integra a una aplicación de consola de Windows. Esta será la aplicación del lado del servidor que se encargará de recibir las conexiones de los clientes, recibir y enviar mensajes a los mismos.

En la *línea 14* del código se hereda de la clase Push, permitiendo así que la aplicación pueda sobrecargar los métodos y se pueda escuchar por nuevas conexiones entrantes.

En las líneas 46 se crea una instancia de la aplicación y en la línea 50 se llama al método Start() de la librería push para comenzar con la ejecución del programa.

```
namespace SMS
{
    class Program:Push.Push
    {
        WebSMSModel.BLL.SMSPhoneModel _phoneModel;
        public override void Initialization()...
        List<ResponseData> ResponseDB;
        List<int> OpenRequests;
        public String Check(int idRequest) ...
        public override string ProcesarComando(string msj)...
        public override void RecibirMensaje(byte[] bytesFrom, string id)...
        void SendSMSAckStatus(String deviceId,int msjId,String status) ...
        TopMost
        static void Main(string[] args)
        {
            TopMost


            Program p = new Program();
            RequestChecker _rc = new RequestChecker(p);
            //_rc.StartCheck();
           // Console.WriteLine("GoOn");
            p.Start();
        }
        class RequestChecker {
            Program p;
            Thread requestCheckerThread;
            public RequestChecker(Program p) ...
            public void StartCheck()...
        }
    }
}

```

Figura 38 – Uso de la librería Push.dll en la aplicación de consola de Windows.

```
Program p = new Program();
p.Start();
```

Código 9 inicialización de servicio

En el método ***ProcessCommand*** se incluye la lógica para los mensajes que serán recibidos desde la página web. Se utiliza el objeto `Message` que tiene en el campo Data el contenido del mensaje de la página web. Para este ejemplo todos los mensajes serán de tipo Unicast, ya que es solo un teléfono que realizará el envío de los mensajes de texto (en un futuro se puede agregar la lógica para que funcione con más teléfonos). En el método *ProcessCommand* se utiliza la función *Send* de la librería y se pasa el mensaje junto con la lista

de dispositivos a los cuales se les hará llegar en mensaje. Una lista con un solo valor para este caso.

En el método ***ReceiveMessage*** se programa la lógica para recibir el estado de los mensajes que fueron enviados desde el teléfono. En esta función se actualiza en Base de datos el estado individual de cada mensaje que debe ser enviado.

4.1.6 Aplicación Cliente

Aplicación java para Android que se encargará de mantener la conexión activa con el servidor utilizando la librería push diseñada para tal fin. Esta librería debe procesar e interpretar los masajes recibidos y permitir el envío de mensajes de texto a través del dispositivo Android cliente. La aplicación debe notificar al servidor push el estado de los mensajes (si fueron enviados, recibidos, o si tuvieron algún error).

4.1.6.1 Modelo de caso de uso

En la Figura 39 se muestra el modelo de caso de uso para el cliente push.

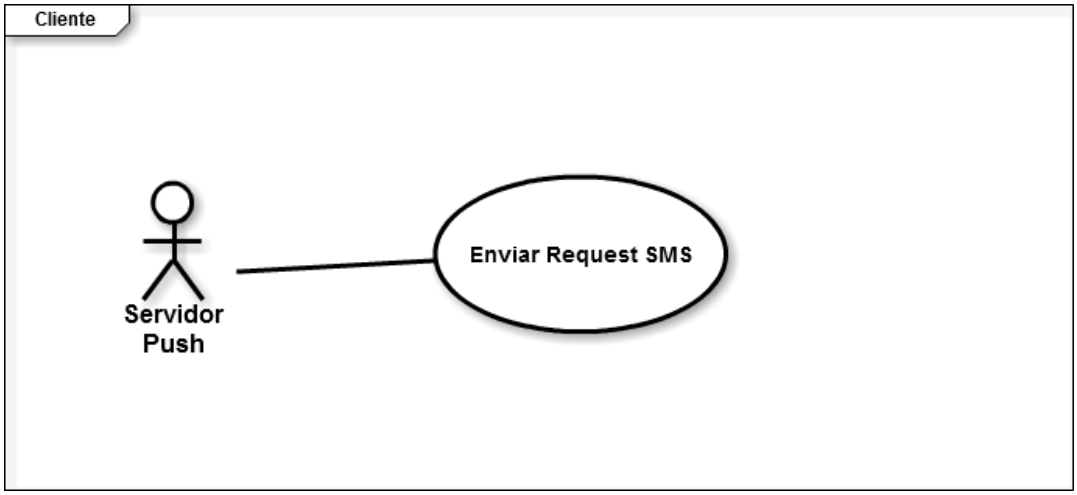


Figura 39 Modelo de caso de uso para cliente push

Enviar Request(Servidor Push): caso de uso que consiste en el envío por parte del servidor push a la aplicación cliente para que esta procese los mensajes que deben ser enviados.

Inicialización del servicio en la actividad principal

La aplicación en Android sólo mostrará en la pantalla el estado de los mensajes recibidos. Mostrará el total de mensajes, los Enviados(S), los no enviados (E) y los que llegaron al dispositivo final(D). En la figura 40 se encuentra la configuración inicial para el servicio. Antes de hacer el llamado se configura la variable *INTENT_FILTER* para que los mensajes sean recibidos por el receiver de la aplicación que tiene como intent filter *ron.tesis.PushService.SMS* que se debe configurar en el *AndroidManifest.* En la Figura 41 se muestra esta configuración en el archivo Manifest. En la Figura 42 se muestra la pantalla resumen de la aplicación Android.

```
public class Inicio extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        Logger.AgregarEntrada(G.TAG,this,"onCreate");
        super.onCreate(savedInstanceState);
        setContentView(R.layout.inicio);

        ron.tesis.PushService.ConnectionService.SetInit(getApplicationContext(),"rk7.co",8890,2,true);
        Intent serviceIntent =new Intent(this,ron.tesis.PushService.ConnectionService.class);
        Bundle extras = new Bundle();
        extras.putString("INTENT_FILTER", "ron.tesis.PushService.SMS");
        serviceIntent.putExtras(extras);
        startService(serviceIntent);

        int [] _res=new DBAccess(this). getSMSResumeCount();
        StringBuilder _resumen= new StringBuilder();
        _resumen.append("TOTAL: ").append(_res[0]+_res[1]+_res[2]+_res[3]).append("\n");
        _resumen.append("S: ").append(_res[1]).append("\n");
        _resumen.append("E: ").append(_res[2]).append("\n");
        _resumen.append("D: ").append(_res[3]);
        ((TextView)findViewById(R.id.resumen)).setText(_resumen.toString());
    }

    public boolean onCreateOptionsMenu(Menu menu) {

}
```

Figura 40 Inicialización del servicio en la actividad principal

```
43        <receiver android:name="ron.tesis.SMSSENDER.PushReceiver">
44         <intent-filter>
45              <action android:name="ron.tesis.PushService.SMS"/>
46         </intent-filter>
47        </receiver>
```

Figura 41 Configuración del receiver en el AndroidManifest

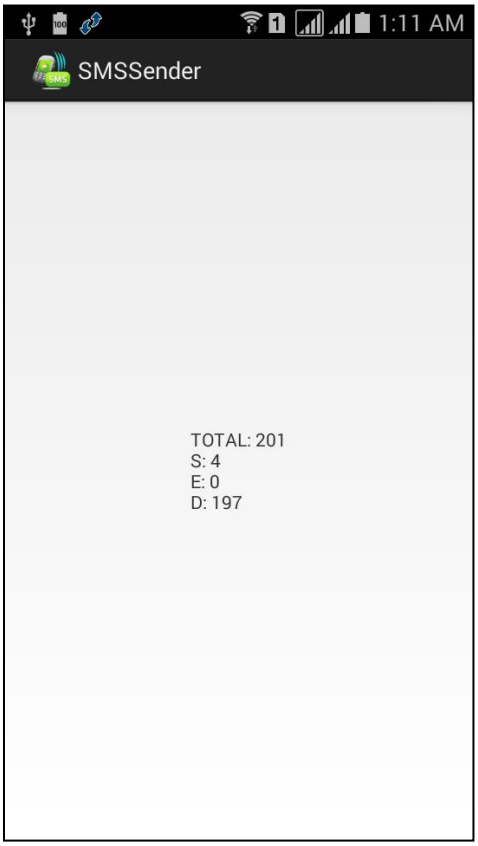


Figura 42 Diseño final cliente Android

4.2 Simple Chat

El segundo caso de estudio es una aplicación simple de chat que no dependerá de ninguna interfaz web. Este chat en Android sólo conectará a todos los dispositivos que instalen la aplicación y permitirá el intercambio de mensajes o imágenes de hasta 500KB entre ellos. La comunicación entre los clientes que se encuentren conectados es completamente pública, es decir podrá ser vista por todos los dispositivos que se encuentren conectados.

Las partes involucradas en este sistema de chat son:

- Servidor Push.
- Cliente(s) Android.

4.2.1 Modelo de caso de Uso

En la Figura 43 se muestra el diagrama de caso de uso del sistema SimpleChat.

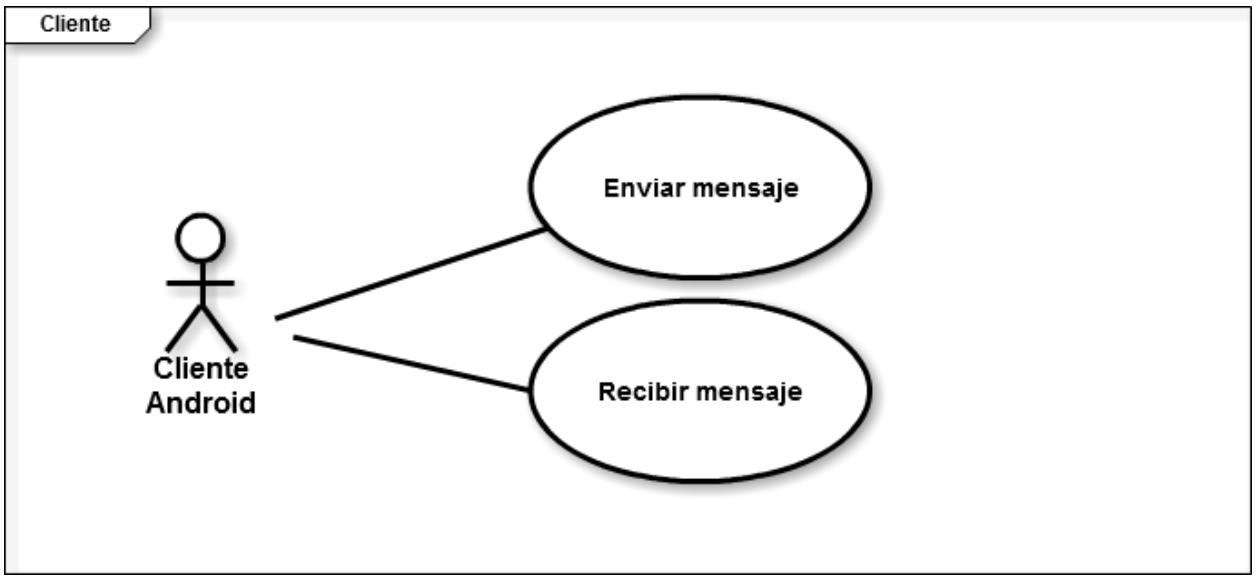


Figura 43 Caso de uso SimpleChat.

Enviar Mensaje: caso de uso que consiste en el envío de mensajes por parte del cliente.

Recibir Mensaje: caso de uso que consiste en la recepción de mensajes por parte del cliente.

4.2.2 Aplicación Servidor

Aplicación o servicio en Windows que se encargará de recibir las peticiones de los dispositivos clientes que se encuentren conectados a la red. La aplicación servidor se encargará de encolar y entregar los mensajes a cada uno de los dispositivos que se encuentren conectados.

Se programa el método *ReceiveMessage* con la lógica para enviar mensajes a todos los dispositivos cada vez que un dispositivo envíe un mensaje. Hay 3 tipos de mensajes que pueden llegar desde el dispositivo cliente con el siguiente formato I;MSJ;EXTRA, en donde I es el identificador del tipo de mensaje, MSJ es el contenido del mensaje, y EXTRA es cualquier información extra que puede ser enviada(Ejemplo, nombre de archivo).

Los Mensajes con el nombre del dispositivo (serán los mensajes cuyo identificador es ‘U’).

Los Mensajes de texto, son mensajes en texto plano cuyo identificador es la letra ‘M’

Las Imágenes, son archivos de imágenes cuyo identificador es la letra ‘F’

Los mensajes que se reciban serán enviados a los dispositivos excepto al que envía el mensaje. Todo haciendo uso de la función pública SendAll(…) en donde se utiliza el parámetro 'but' para excluir el cliente actual. En la Figura 44 se muestra el código utilizado para la creación del servidor. Los mensajes y archivos serán codificados a un String en Base64 al ser enviado al servidor. El cliente al recibir el mensaje debe decodificar de nuevo esta información.

```
10      class Program : Push.Push
11      {
12
13          public override void ReceiveMessage(long msjId,byte[] bytesFrom, string id)
14          {
15
16              String shortId = id.Length > 8 ? id.Substring(0, 8) : id;
17
18              String dataFromClient = System.Text.Encoding.UTF8.GetString(bytesFrom);
19
20              Console.WriteLine("{0} <<envió {1}", shortId, dataFromClient);
21
22              if (dataFromClient[0] == 'U')
23              {
24                  String[] _splt = dataFromClient.Split(new char[] { ';' }, 2);
25                  Connections[id].Extra = _splt[1];
26                  new ChatBLL().UpdateDevice(id, _splt[1]);
27              }
28              else if (dataFromClient[0] == 'M')
29              {
30                  String name="";
31                  if (Connections[id].Extra != null)
32                      name = Connections[id].Extra.ToString();
33                  String[] _splt = dataFromClient.Split(new char[] { ';' }, 2);
34                   SendAll(id,msjId,"M;" + name +";"+ _splt[1], true, id);
35              }
36              else if (dataFromClient[0] == 'F')
37              {
38                  String name = "";
39                  if (Connections[id].Extra != null)
40                      name = Connections[id].Extra.ToString();
41                  String[] _splt = dataFromClient.Split(new char[] { ';' }, 3);
42                  SendAll(id,msjId,dataFromClient+";"+name, true, id);
43              }
44          }
45          static void Main(string[] args)
46          {
47
48              Program p = new Program();
49              p.Start();
50          }
51      }
```

Figura 44 Servidor SimpleChat

4.2.3 Aplicación Cliente

La aplicación cliente sólo consistirá en una interfaz sencilla en donde se almacenen y muestren los mensajes que se reciban de los dispositivos que se encuentren conectados. Con un campo de texto dos botones, un botón para el envío de mensajes y un botón adicional que permitirá la selección de algún archivo de imagen no mayor a 100KB para que pueda ser enviado Figura 45.

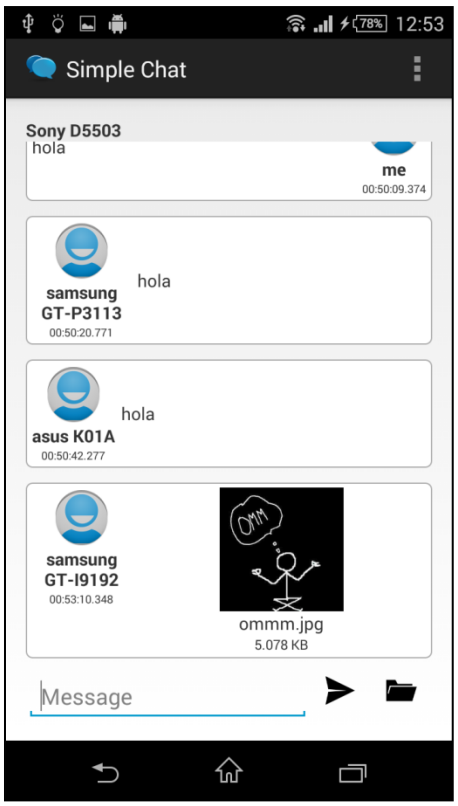


Figura 45 Aplicación cliente SimpleChat

4.3 Twitter Push

El tercer caso de estudio consiste en una aplicación que tiene como función notificar via push o mensaje de texto información sobre algún tema en Twitter u obtener actualizaciones del timeline de algún usuario en particular. Para esto se hará uso de las aplicaciones desarrolladas previamente. El envío de los mensajes de texto se hará utilizando el servidor push de la aplicación WebSMS y el envío de las notificaciones se hará a través del servidor SimpleChat. Se debe crear un portal web en donde el usuario debe registrarse, suscribirse a los temas de

interés e indicar como quiere que lleguen esas actualizaciones (Vía sms o vía push). También se debe desarrollar la aplicación que estará monitoreando Twitter para obtener los temas deseados por los usuarios. Esta misma aplicación será la encargada de comunicarse con el servidor SimpleChat o WebSMS para que estas sean las encargadas del arribo de los mensajes.

Las partes involucradas en este sistema son:

- Página web de registro y suscripción de temas.
- Aplicación Monitor de Twitter.
- Servidor de envío de SMS (Elaborado).
- Servidor envío de mensajes push (Elaborado).

4.3.1 Modelo de caso de Uso

En la Figura 46 se muestra el diagrama de caso de uso del sistema.

Registrarse: caso de uso que consiste en el registro del usuario introduciendo sus datos y creando su cuenta para poder acceder al sistema.

Iniciar Sesión: caso de uso que consiste el inicio de sesión del usuario al sistema.

Configurar Temas: caso de uso que consiste en la selección y configuración de temas de interés del usuario.

Registrar equipo: caso de uso que consiste en el registro del equipo utilizando la cuenta de usuario creada para la recepción de los mensajes vía push utilizando la aplicación SimpleChat.

Recibir SMS: caso de uso que consiste en la recepción de mensajes de texto por parte del usuario con actualizaciones del tema seleccionado.

Recibir Push: caso de uso que consiste en la recepción de mensajes vía push por parte del usuario con actualizaciones del tema seleccionado a través de la aplicación SimpleChat.

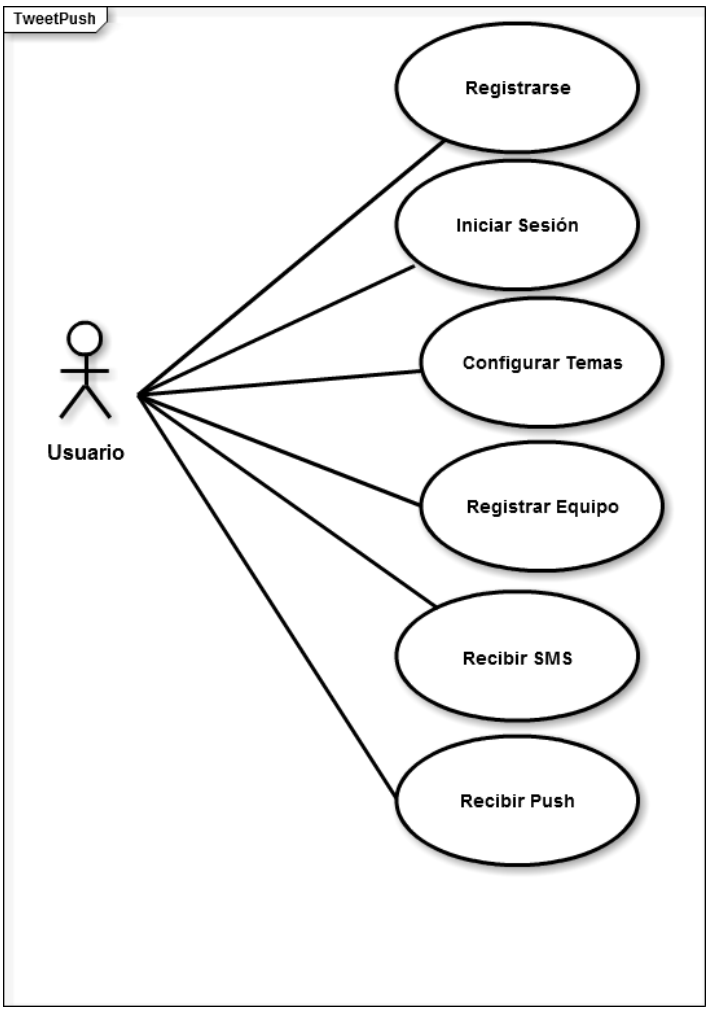


Figura 46 Caso de Uso TweetPush

4.3.2 Diagrama de secuencia

En la Figura 47 se muestra el Diagrama de secuencia del sistema.

1.1.1 Base de Datos

Para el funcionamiento del portal y del monitor web se debe crear una Base de datos con las tablas necesarias para almacenar la configuración del usuario. Como el sistema planteado aprovechará los servicios push creados anteriormente por lo que no es necesario la importación del script para crear las tablas asociadas a la mensajería push. En su lugar se crean tres tablas sencillas para almacenar los datos y las preferencias del usuario. Las tablas necesarias y la relación entre ellas se muestran en la Figura 48.

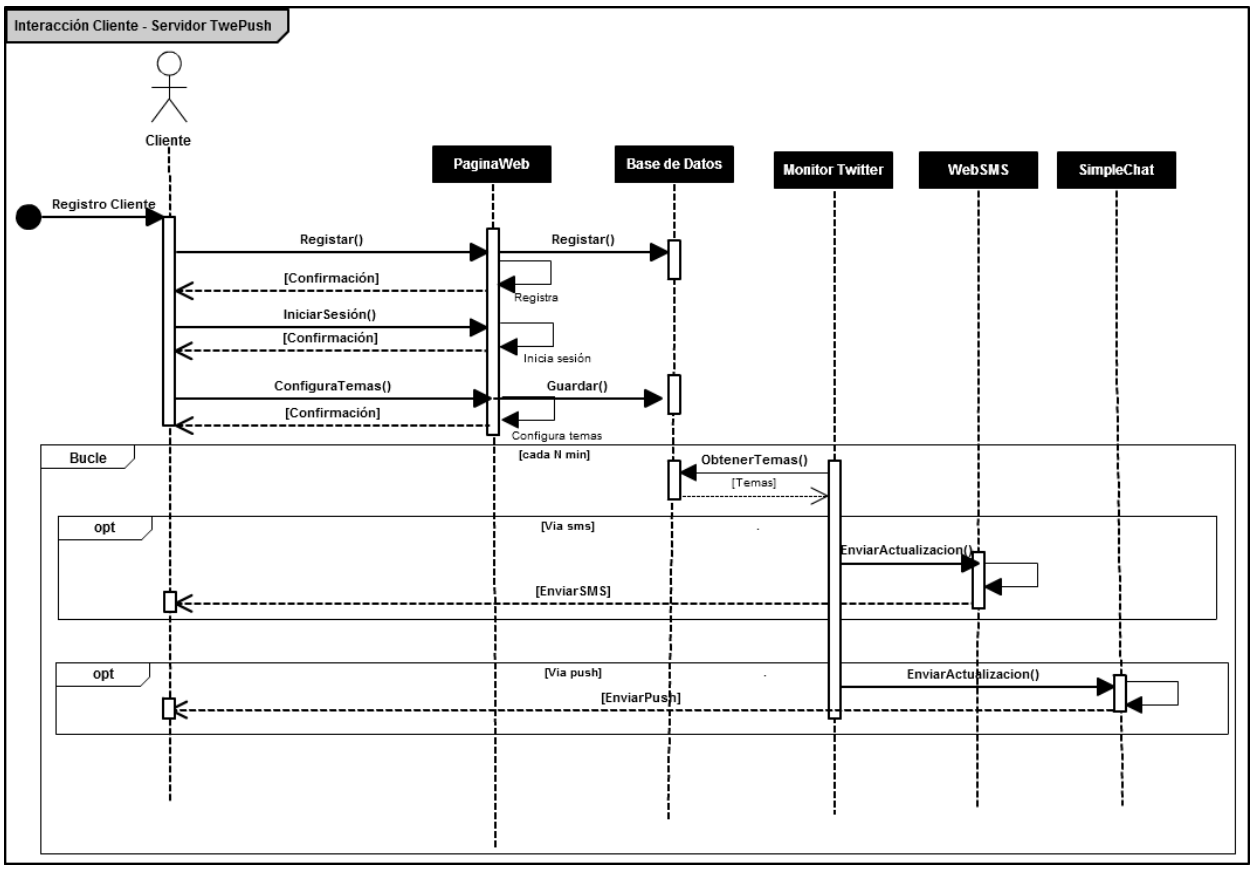
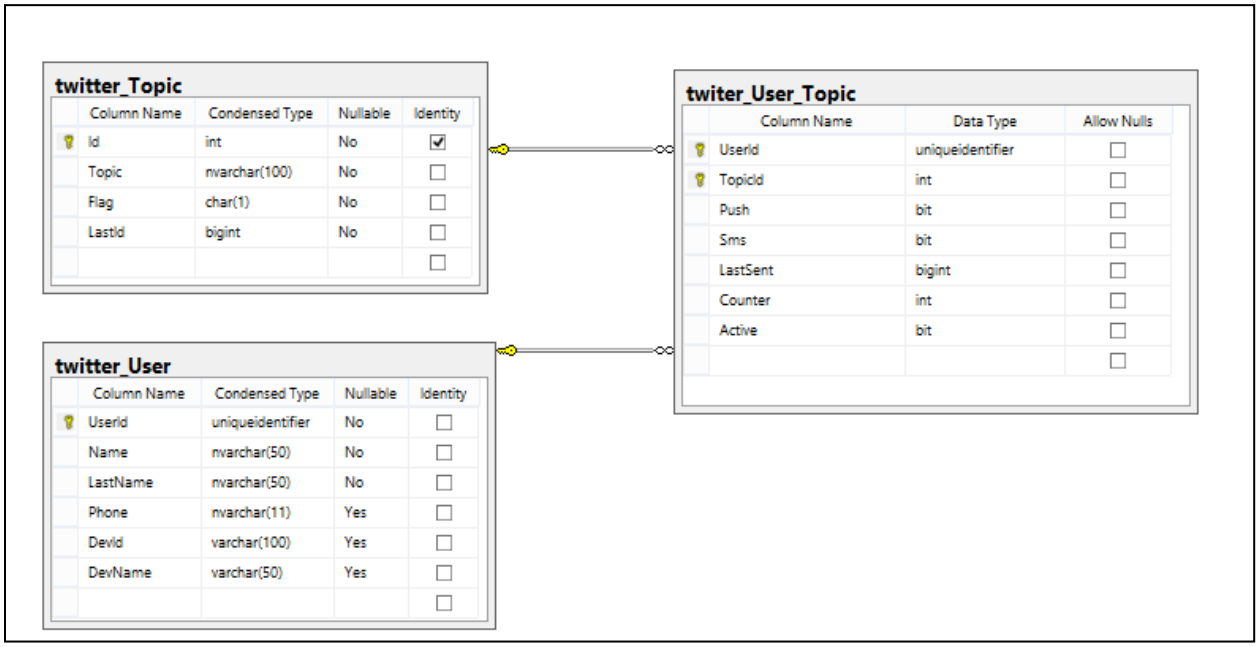


Figura 47 Diagrama de secuencia TweetPush.



twitter_Topic

Column Name	Condensed Type	Nullable	Identity
Id	int	No	☑
Topic	nvarchar(100)	No	☐
Flag	char(1)	No	☐
LastId	bigint	No	☐
			☐

twiter_User_Topic

Column Name	Data Type	Allow Nulls
UserId	uniqueidentifier	☐
TopicId	int	☐
Push	bit	☐
Sms	bit	☐
LastSent	bigint	☐
Counter	int	☐
Active	bit	☐
		☐

twitter_User

Column Name	Condensed Type	Nullable	Identity
UserId	uniqueidentifier	No	☐
Name	nvarchar(50)	No	☐
LastName	nvarchar(50)	No	☐
Phone	nvarchar(11)	Yes	☐
DevId	varchar(100)	Yes	☐
DevName	varchar(50)	Yes	☐
			☐

Figura 48 Tablas TweetPush.

twitter_User: contiene datos asociados al perfil del usuario y datos importantes para el envío de los mensajes, ya sean via push o sms. El campo **(UserId)** es el Id asociado al usuario, el nombre y el apellido se almacenan en los campos **(Name)** y **(LastName)** respectivamente, el teléfono celular que será utilizado para el envío de mensajes de texto es almacenado en el campo **(Phone),** mientras el Id del dispositivo para el envió de mensajes push se almacena en el campo **(DevId).** El campo **(DevName)** es utilizado para almacenar el nombre (marca/modelo) del teléfono registrado.

twitter_Topic: tabla que contiene los temas de interés que han sido agregado por los usuarios en el sistema. Los temas que se agreguen serán visibles por todos los usuarios que estén registrados para que puedan agregarlos a sus preferencias. Los campos de esta tabla incluyen un id único **(ID),** el nombre del tema **(Topic),** un flag **(Flag)** que indicará si se trata de el timeline de un usuario o si es una búsqueda con hashtag, y un campo **(LastId)** que indicará cuál fue el último tweet que se obtuvo durante la última búsqueda.

twitter_User_Topic: tabla que será utilizada para almacenar la relación entre los temas existentes y el usuario que desea obtener actualizaciones de dicho tema. La clave primaria de esta tabla son el **(UserId)** y el **(TopicId)** perteneciente a los Id de usuario y del tema de interés respectivamente. El campo **(Push)** y **(Sms)** indicarán si se desea recibir las notificaciones del tema vía push o vía mensaje de texto, el campo **(LastSent)** almacenará el Id del ultimo tweet que se le envió al usuario, el campo **(Counter)** irá contando la cantidad de tweets que se envían, y **(Active)** indica si el tema está habilitado para que sea enviado al usuario o no (Activado/desactivado).

1.1.2 Portal Web

Página web que se encargará del registro del usuario y de proporcionar una interfaz amigable para la suscripción/de suscripción a temas de interés por twitter.

1.1.2.1 Intefaz del portal web

En esta sección se muestra la interfaz del portal web.

Home (Inicio)

Pantalla principal del sistema TweetPush que muestra los enlaces para iniciar sesión o crear una cuenta nueva. En la Figura 49 se muestra la pantalla principal de sistema.

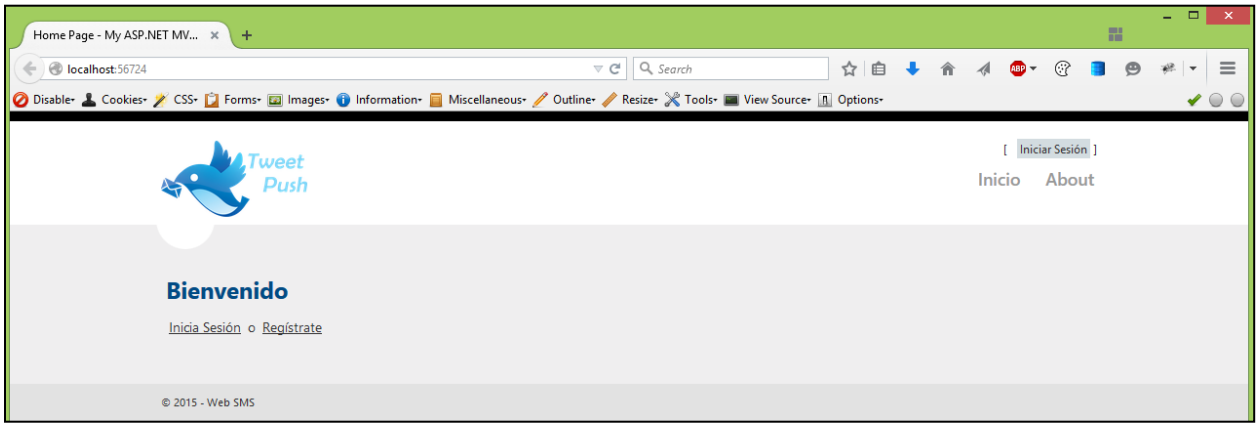


Figura 49 Página principal TweetPush

Registro

Pantalla que muestra el formulario de registro para el sistema TweetPush. El usuario debe rellenar el formulario y presionar el botón crear para crear una cuenta nueva. En la Figura 50 se muestra la pantalla de registro del sistema.

Inicio Sesión

Pantalla que muestra el formulario de inicio de sesión del sistema TweetPush. El usuario debe ingresar los datos requeridos para iniciar sesión en el sistema. En la Figura 51 se muestra la pantalla de inicio de sesión.

Pantalla de suscripción

Pantalla de suscripción y configuración del los temas de interés. En esta pantalla se agregan temas de la lista de temas existente o se agregan nuevos temas en caso de que no existan. Las opciones Mention y TL indican si el tema de interés debe ser buscado como una mención utilizando hashtags o si se trata del timeline de un usuario. Los checkbox Notification y SMS indican por cuál vía deben enviarse las notificaciones. Y el campo Enable indica si se debe tomar en cuenta el tema o no (si está habilitado o deshabilitado). Al agregar o eliminar temas se

debe presionar el botón Actualizar suscripción para que los datos sean almacenados. En la Figura 52 se muestra la pantalla de suscripción.

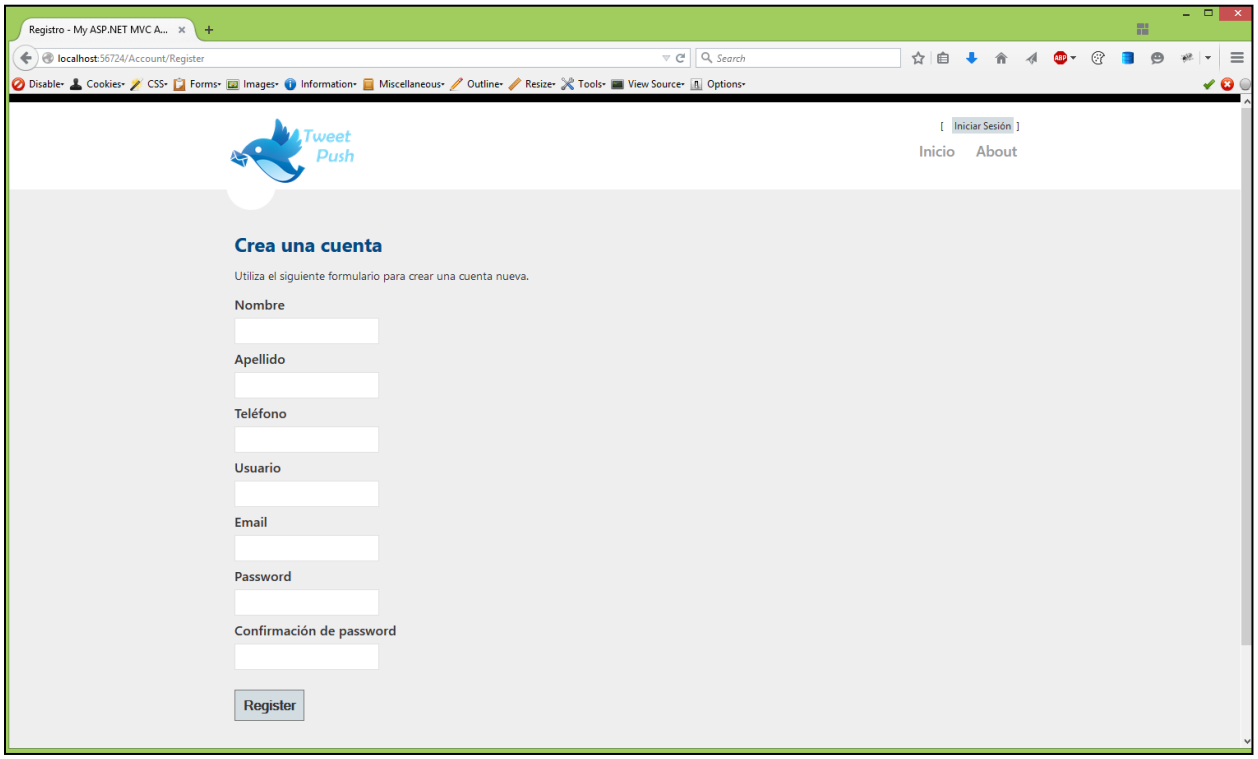


Figura 50 Página de Registro TweetPush

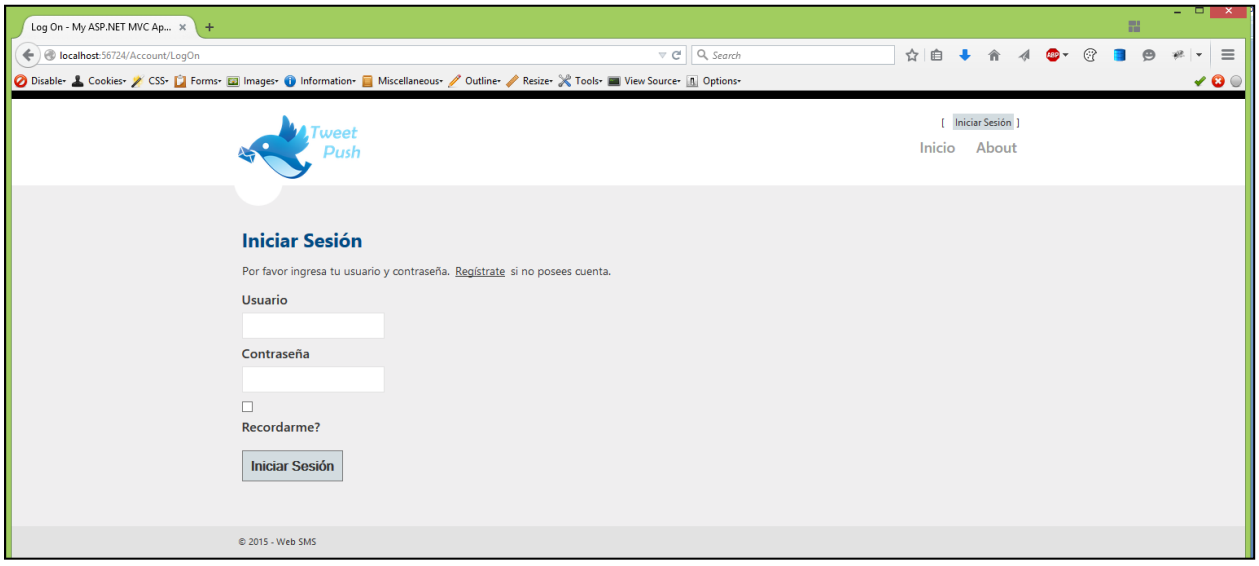


Figura 51 Página de Inicio de sesión TweetPush

Figura 52 pantalla de suscripción TweetPush

1.1.2.2 API de registro de dispositivo

Adicionalmente se desarrollo dentro del mismo portal un método que actúe como web api para asociar un dispositivo utilizando la aplicación SimpleChat al usuario del sitio web TweetPush.

1.1.3 Monitor Twitter

Aplicación de Windows que se encargará de buscar en Base de datos los temas de interés que han agregado los usuarios y enviar los últimos tweets de los temas ya sea vía mensaje de texto o via push. La aplicación debe conocer la dirección y los puertos por donde están escuchando los respectivos servicios y debe enviar las peticiones a través del puerto de comandos. Para realizar la búsqueda de tweets se hará uso de una librería diseñada para ese fin *Tweetinvi*[9].

Adicionalmente la aplicación de servidor SimpleChat debe ser modificada para que procese los mensajes por el puerto de comandos y se encargue de direccionar los mensajes a los dispositivos correspondientes utilizando el objeto Mensaje de la y definiendo el tipo de envío como Multicas. En la Figura 53 se muestra el método ProcessCommand implementado.

[9] https://tweetinvi.codeplex.com/

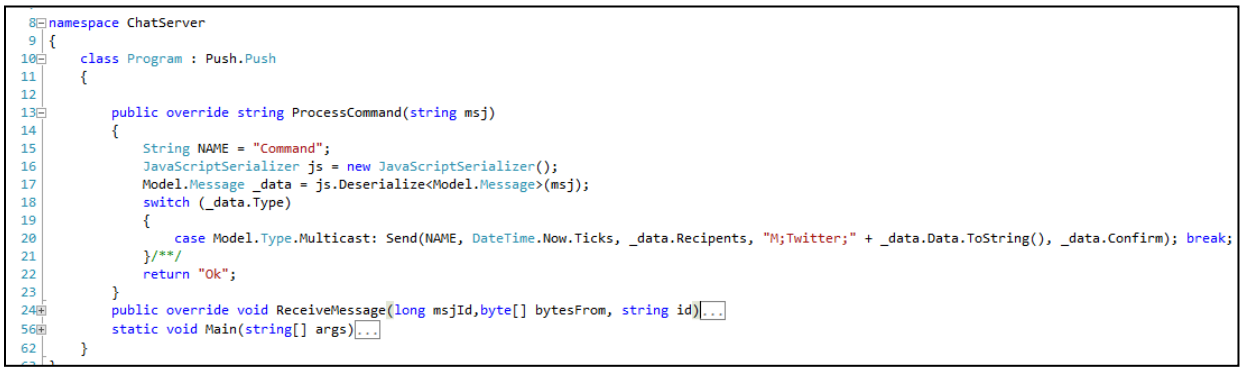


Figura 53 Método ProcessCommand en la aplicación servidor SimpleChat.

1.1.4 Aplicación Cliente

La aplicación cliente en este caso será la misma aplicación creada para el sistema SimpleChat, pero se debe agregar la opción de registro del dispositivo para que el Id quede registrado en la Base de Datos asociado al usuario. En la Figura 54 se muestra la opción agregada en la aplicación existente SimpleChat. En caso de haber configurado el tema para recibir notificaciones vía push estas llegarán a través de la aplicación SimpleChat en caso de haber configurado el tema para recibir las notificaciones vía mensajes de texto estas llegarán al dispositivo destino vía sms. En la Figura 55 se muestran las notificaciones recibidas vía push(izq) y vía sms(der) a través de la aplicación SimpleChat.

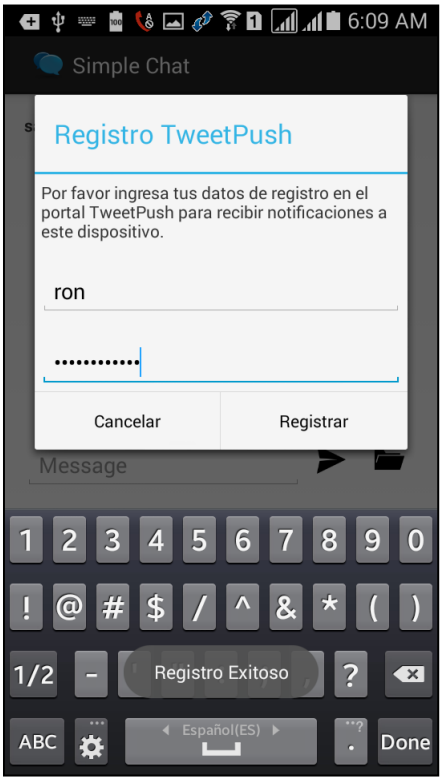


Figura 54 Cuadro de registro en aplicación SimpleChat.

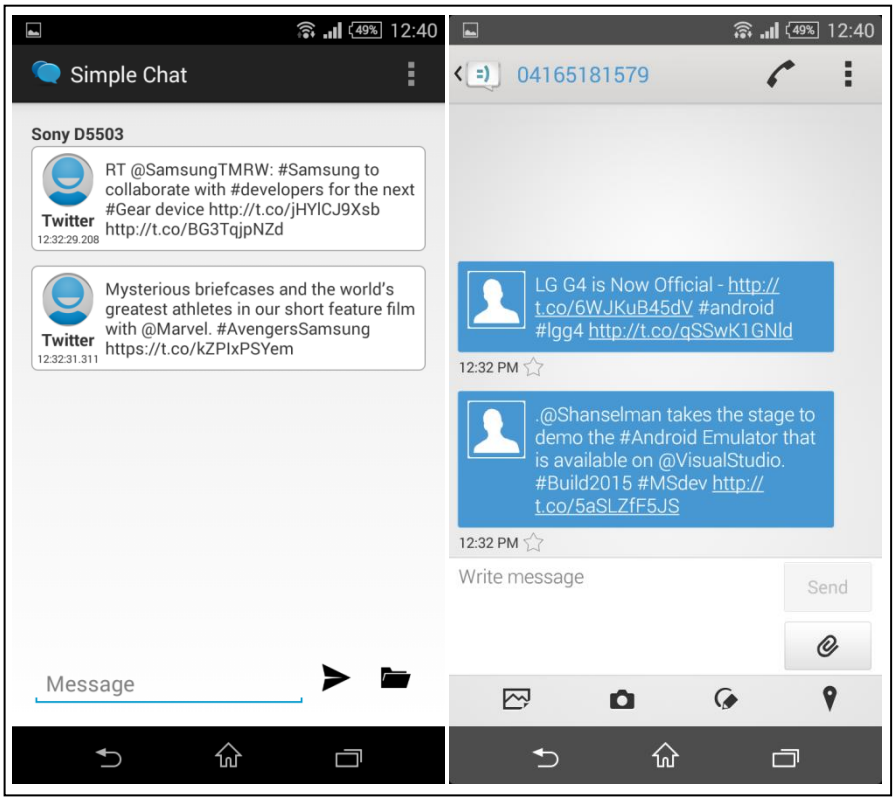


Figura 55 Notificaciones recibidas.

Capítulo 5: Pruebas y Resultados

En este capítulo se realizarán pruebas sobre las aplicaciones diseñadas para obtener una conclusión sobre el rendimiento, capacidad, robustez y fiabilidad de las librerías.

Los dispositivos Clientes utilizados para las pruebas son:

- *Samsung Galaxy I9192 (Android 4.2.2) [I9192]*
- *Sony Z1 Compact D5503 (Android 4.4.4) [Sony]*
- *Galaxy tab 2 P3113 (Android 4.2.2) [P3113]*
- *Tablet Asus K01A (Android 4.4.2) [Asus]*

Los dispositivos habilitados para el uso de datos (EDGE/3G/HSDPA) y envío de mensajes de texto son el *I9192* y el *Sony.* Ambos bajo la Operadora Movilnet.

Todos los dispositivos cuentan con interfaces WiFi.

El Servidor a utilizar será un computador con *Windows Server 2008 R2 Enterprise*, que posee como Sistema Manejador de Base de Datos Microsoft SQL Server 2012, IIS 7 como servidor web para hospedar las aplicaciones web.

Las pruebas y las aplicaciones desarrolladas fueron creadas con la intención de mostrar el funcionamiento y las capacidades de las librerías por lo que se omitieron detalles y funcionalidades que deben ser implementadas si se piensan utilizar en un ambiente real de producción.

Para las prueba de envío de SMS se utilizó el dispositivo *Sony*. Para las pruebas con la aplicación SimpleChat se utilizaron los dispositivos *I9192, Sony, P3113 y Asus.* Para la prueba con la aplicación TweetPush se utilizó el dispositivo *Sony.* La arquitectura del sistema para las pruebas se puede observar en la Figura 56.

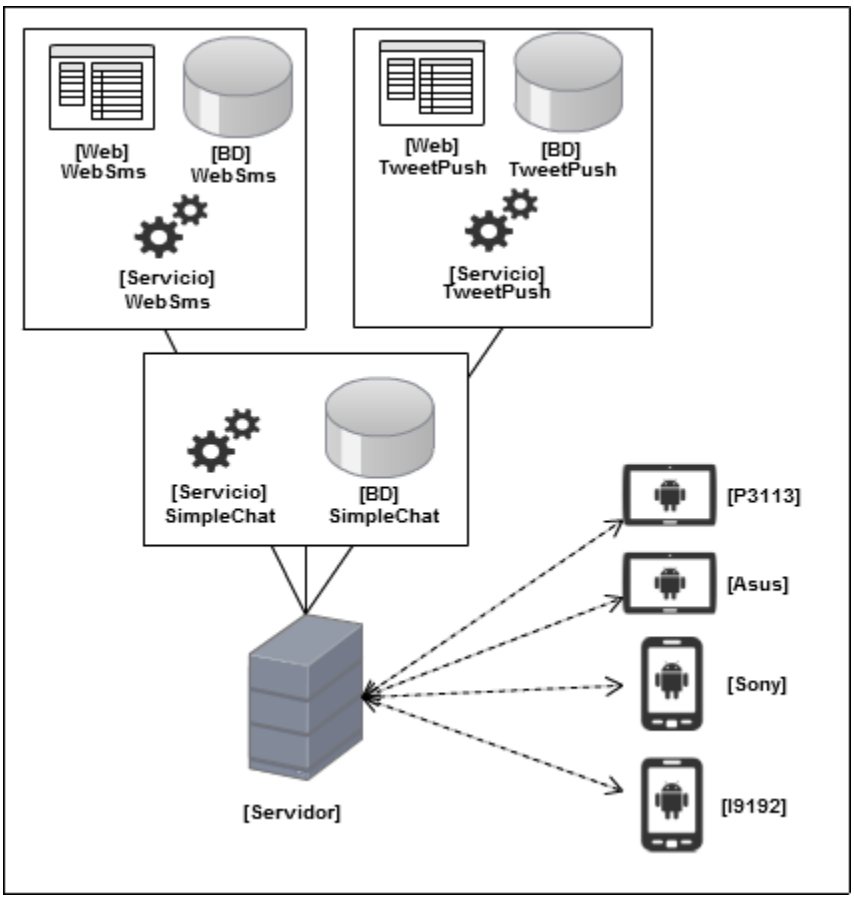


Figura 56 Arquitectura del Sistema para las pruebas

5.1 Web SMS

En esta aplicación se utilizará un teléfono *Sony Z1 Compact D5503* como dispositivo cliente para el envío de los mensajes de texto.

5.1.1 Prueba 1 [Envío de múltiples mensajes].

La prueba consiste en el envío de múltiples mensajes a teléfonos celulares con Línea Movilnet. Utilizando los dispositivos I9192 (GSM), Sony (GSM,DualSim), LG MX380(CDMA).

Las pruebas se dividirán en 3 Fases:

En la primera fase se enviará un mensaje con una longitud de 25 caracteres a cuatro números telefónicos (4 mensajes).

En la segunda fase se enviará un mensaje de 40 caracteres a cuatro números telefónicos, cada número se encuentra en la lista dos veces. (8 mensajes).

En la tercera Fase se enviará un mensaje de 50 caracteres a cuatro números telefónicos, cada número se encuentra en la lista diez veces. (40 mensajes).

En total se crearán 3 peticiones (request) de envió de mensajes a través de la página. Los mensajes se repetirán en los dispositivos receptores de acuerdo a la cantidad de veces que el número telefónico aparezca en la lista. El total de mensajes de texto enviados es 52 (4 en la Fase 1, 8 en la Fase 2 y 40 en la Fase3)

Fase 1:

En la Figura 57, se crea el mensaje y se agregan los números destinos (separados por coma) que recibirán el mensaje *"Mensaje Prueba 1\n4numeros"* los mensajes serán enviados sin interrumpir la comunicación. En la Figura 58 se ha creado y enviado el request exitosamente. En la Figura 59 se observa en la misma pantalla actualizada en la que todos los mensajes llegaron exitosamente sin ningún contratiempo.

Fase 2:

En esta fase se crea el mensaje en la página de creación de mensaje, se agregan los números destinos (separados por coma y repetidos dos veces) que recibirán el mensaje *"*Mensaje de prueba 2\n8 números 2 cada uno*"* los mensajes serán enviados sin interrumpir la comunicación. En la Figura 60 se ha creado y enviado el request exitosamente.

Fase 3:

En esta fase se crea el mensaje en la página de creación de mensaje, se agregan los números destinos (separados por coma y repetidos dos veces) que recibirán el mensaje *"*Prueba 3 40 Mensajes 4\nnumeros x 10 veces cada uno*"* los mensajes serán enviados sin interrumpir la comunicación. En la Figura 61 se ha creado y enviado el request exitosamente.

Como la aplicación servidor se encarga de verificar cada cierto tiempo la cantidad de requests pendientes a ser enviado y no los asigna (encola) a un dispositivo en específico. Pruebas de desconexión no son relevantes en este caso ya que el servicio siempre enviará el mensaje push con el request al dispositivo que esté configurado en el webconfig (Siempre que se encuentre conectado).

Figura 57 Creación de Request para envío de mensaje de texto.

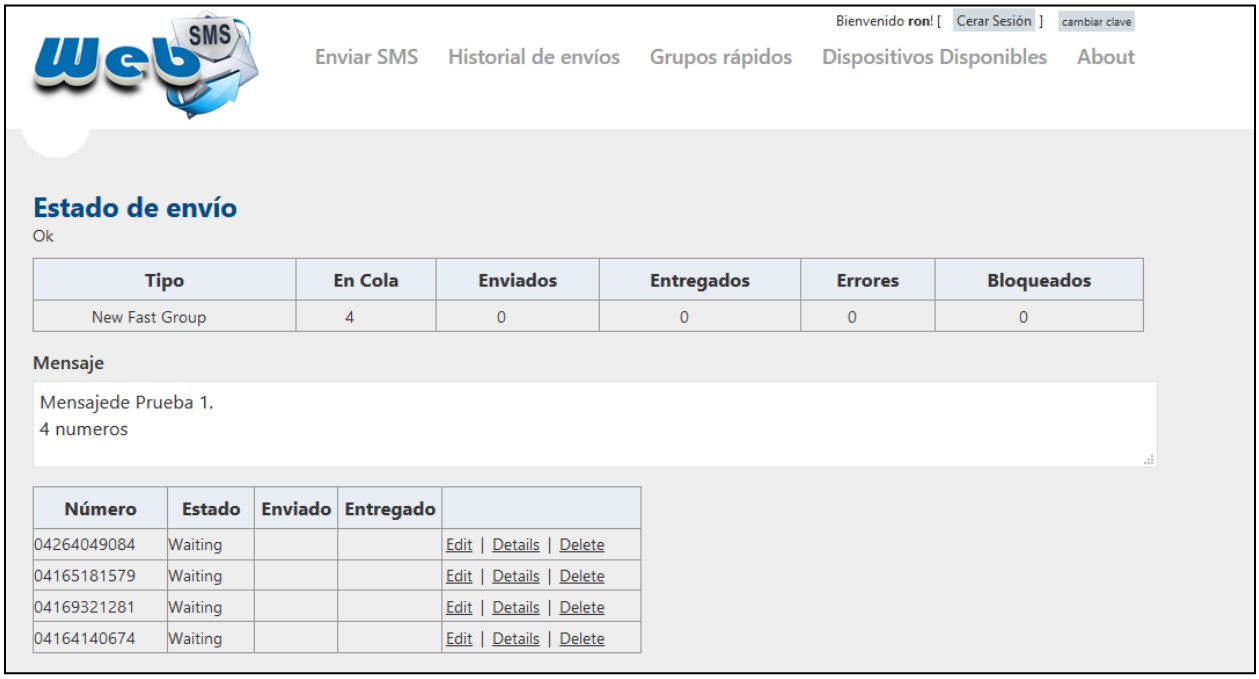


Figura 58 Creación de Request satisfactorio.

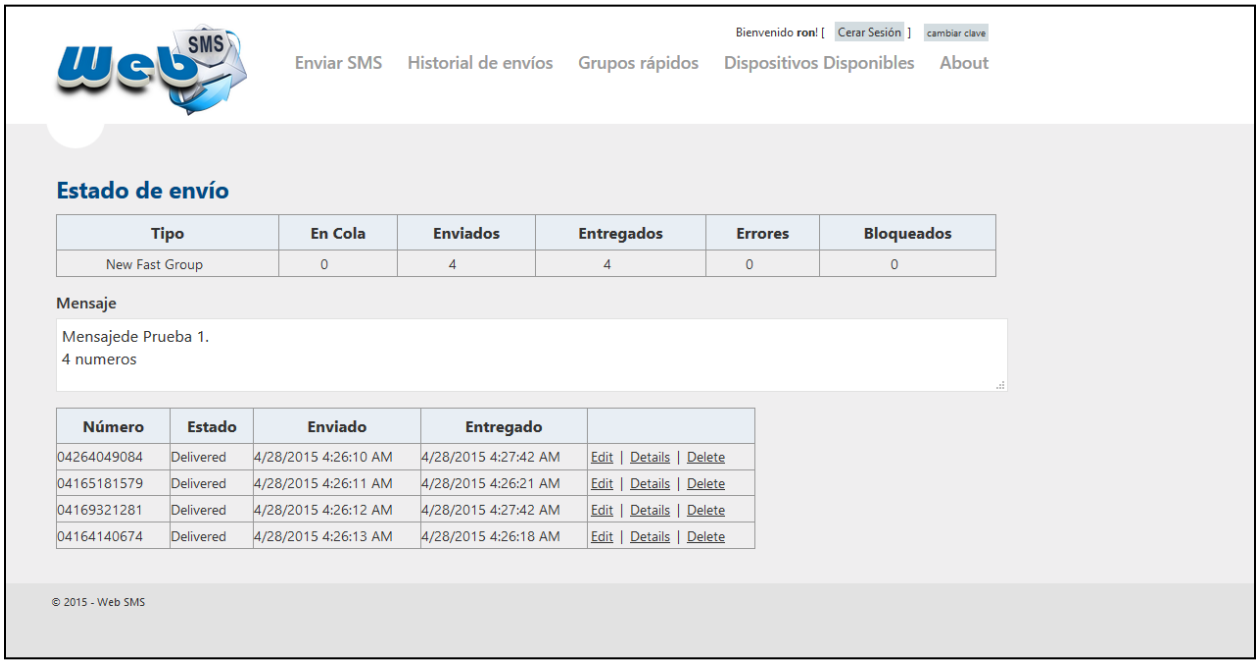


Figura 59 Resultado Fase 1

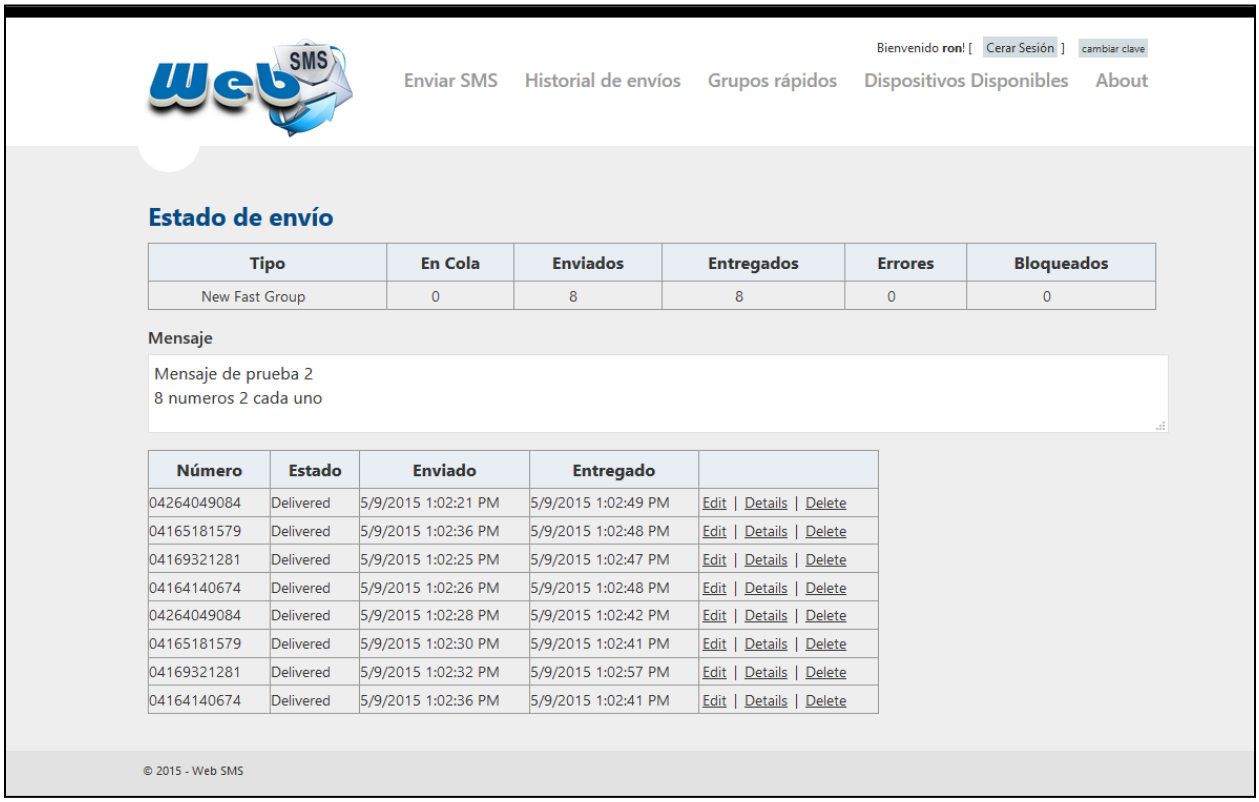


Figura 60 Resultado Fase 2

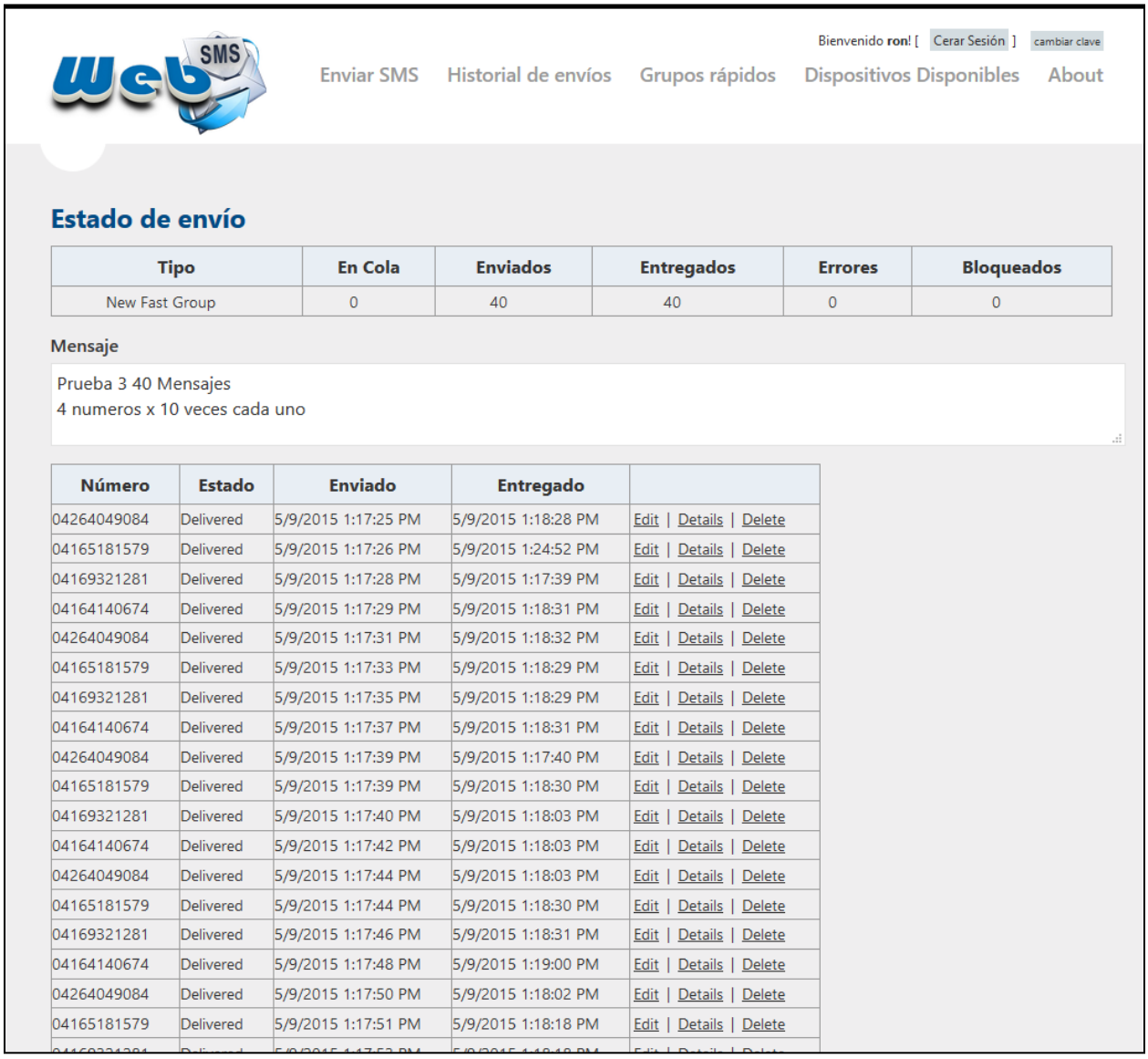


Figura 61 Resultado Fase 3

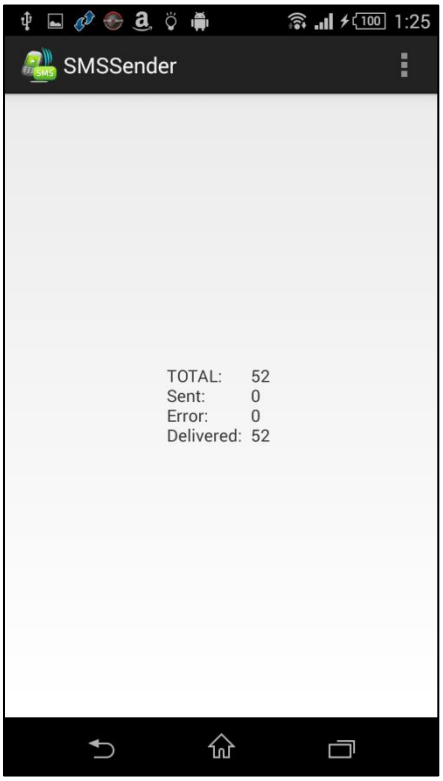


Figura 62 Estado de mensajes en aplicación cliente.

Resultados

Los mensajes de request llegaron al teléfono exitosamente y los mensajes fueron enviados. Para la Fase 3 de la prueba el primer mensaje de texto se envió a la 1:17:25pm el último mensaje fue enviado a la 1:18:23pm para una diferencia de. El último mensaje fue recibido a la 1:24:52pm.

En la Figura 62 se observa el Resumen de los mensajes procesados por la aplicación cliente.

5.2 SimpleChat

En esta aplicación se utilizarán los dispositivos: Samsung Galaxy I9192, Sony Z1 Compact D5503, Galaxy tab 2 P3113, Tablet Asus K01A.

5.2.1 Prueba 1 [Envío y recepción de mensajes Online].

La primera prueba consiste en el envío de mensajes desde cada uno de los dispositivos mientras estuvieran conectados a la red. Se enviaron 25 mensajes con el texto Hola Mundo N

(12-13 bytes) desde cada dispositivo en donde N es un número entre 1 y 25 que indica el numero de mensaje enviado. Los dispositivos I9192, Asus, P3113 estarán conectados vía WiFi. E dispositivo Sony estará conectado a través de la red de datos de Movilnet. Los mensajes se enviaran con intervalos de 1 segundo entre cada dispositivo e intervalos de 2 minutos entre cada serie nueva de envíos.

Resultado

Todos los dispositivos recibieron los mensajes de forma correcta y de forma instantánea.

En la Figura 63 se muestra la pantalla final del dispositivo Sony con los mensajes recibidos y enviados.

5.2.2 Prueba 2 [Envío y recepción de mensajes Offline].

La segunda prueba consiste en el envío de 30 mensajes hacia y desde los dispositivos que se encuentran conectados y no conectados. La prueba se divide en dos fases:

- **Fase 1:** Envío de mensajes con los dispositivos *P3113 y D5503* desconectados y los dispositivos I9192 y Asus conectados via WiFi.
- **Fase 2:** Se procederá a conectar nuevamente los dispositivos *P3113 (WiFi)* y *Sony (Datos)* luego de 6 minutos y se observarán los resultados.

El envío de los mensajes de la fase 1 se realizará en el siguiente orden:

1) I9192:	Mensaje 1	11) P3113:	Mensaje 11	21) I9192:	Mensaje 21
2) P3113:	Mensaje 2	12) I9192:	Mensaje 12	22) P3113:	Mensaje 22
3) Asus:	Mensaje 3	13) Asus:	Mensaje 13	23) Asus:	Mensaje 23
4) Sony:	Mensaje 4	14) Sony:	Mensaje 14	24) Sony:	Mensaje 24
5) Sony:	Mensaje 5	15) P3113:	Mensaje 15	25) I9192:	Mensaje 25
6) Sony:	Mensaje 6	16) Sony:	Mensaje 16	26) Asus:	Mensaje 26
7) P3113:	Mensaje 7	17) P3113:	Mensaje 17	27) P3113:	Mensaje 27
8) I9192:	Mensaje 8	18) Asus:	Mensaje 18	28) I9192:	Mensaje 28
9) I9192	Mensaje 9	19) I9192	Mensaje 19	29) Sony	Mensaje 29
10) Asus:	Mensaje 10	20) Asus:	Mensaje 20	30) P3113:	Mensaje 30

Resultados

Durante la Fase 1 sólo los dispositivos que se encontraban conectados (*I9192* y *Asus)* recibieron los mensajes entre sí. Mientras que los dispositivos no conectados (*Sony y P3113)* mantienen los mensajes encolados a la espera del restablecimiento de la conexión. El estado final de los dispositivos al finalizar la Fase uno se puede observar en la Figura 64 y la Figura 65.

Durante la Fase 2, luego de conectar los dispositivos (*Sony y P3113)* se realizó el envío y recepción de mensajes. El primer dispositivo en conectar fue el *Sony* y por lo tanto fue el primero en recibir los mensajes de los dispositivos (*I9192 y Asus)* y enviar sus respectivos mensajes encolados que fueron recibidos por *I9192 y Asus*. Cuando el dispositivo *P3113* finalmente se conectó recibió los mensajes enviados por todos los dispositivos (en el orden en que llegaron al servidor) y envió sus respectivos mensajes encolados que fueron recibidos por los otros dispositivos que ya estaban conectados. El estado final de los mensajes se puede apreciar en la Figura 66 y Figura 67.

Nota: La aplicación no reordena los mensajes. Los mismos se imprimen en el orden en que se enviaron/llegaron al dispositivo. Lo ideal para una aplicación real es ordenar en base a un campo "hora recibida en el servidor".

5.2.3 Prueba 3 [Envío de alto volumen de datos].

La tercera prueba consiste en el envío de alto volúmenes de datos por mensaje. Para esto se enviarán imágenes a los dispositivos para observar el comportamiento de la librería con un alto volumen de datos. Se envió un total de 9 mensajes que consisten en imágenes distintos tamaños. El tamaño de las imágenes irá incrementando desde los ~100KB a los ~900KB. La primera imagen será de ~100KB, la segunda de ~200KB, la tercera, y así sucesivamente hasta llegar a la novena imagen que tendrá un tamaño aproximado de 900KB.

Resultado

Se enviaron imágenes de distintos tamaños desde 100KB hasta 900KB y todos los dispositivos recibieron sin problema las imágenes inferiores a 500KB. Las imágenes de tamaño superior llegaron pero al intentar almacenar y obtener el contenido en la base de datos SQLite (*receivedmessages*) ocurría un error debido al volumen de los datos almacenados. En ocasiones

el dispositivo recibía la imagen sin problema. Se hicieron cambios y pruebas almacenando el contenido como Preferencias compartidas (*SharedPreferences*) y se mejoró un poco el error. Sin embargo ante grandes volúmenes de datos la aplicación arrojaba errores de falta de memoria (*Out-of Memory*) el dispositivo que presentaba la falla más frecuentemente es el *P3113* que de todos los dispositivos es el que tiene menores prestaciones.

Nota: el contenido enviado al servidor es más grande que el tamaño original de la imagen, ya que también se envían otros parámetros como id de mensaje, headers, etc. Antes de ser enviada, la imagen se convierte en un arreglo de bytes y este arreglo es codificado como un String en Base64 por lo que el tamaño suele ser un poco mayor al de la imagen original. Una imagen de 400KB puede llegar a generar un payload de ~550KB durante la transmisión.

5.2.4 Prueba 4 [Interrupción de conexión durante la recepción de datos].

La cuarta prueba consiste en el envío de datos imagen 361.834 KB ~494076 bytes a un dispositivo e interrumpir la comunicación varias veces durante la recepción del mensaje para observar cómo se maneja la retransmisión de los datos. Los dispositivos a ser utilizados en esta prueba son *Sony(receptor), I9192(emisor)* . El teléfono receptor utilizará la interfaz de datos móvil para la recepción del mensaje.

Resultado

La recepción de los datos fue exitosa. La librería pudo manejar correctamente el reenvío de mensajes con gran volumen de datos tras la desconexión durante la recepción del mensaje. El resultado se aprecia en la Figura 68.

5.3 TweetPush

5.3.1 Prueba 1 [Recepción de mensajes push y sms].

La Prueba consiste en suscribirse a temas específicos y ejecutar la aplicación servidor Tweet Push mientras los dispositivos se encuentran desconectados para validar la recepción de las actualizaciones. En esta aplicación se utilizarán los dispositivos: *Samsung Galaxy I9192, Sony Z1 Compact D5503.*

Resultado

Los mensajes push llegaron al momento de conectar los dispositivos y los mensajes de texto llegaron casi al instante.

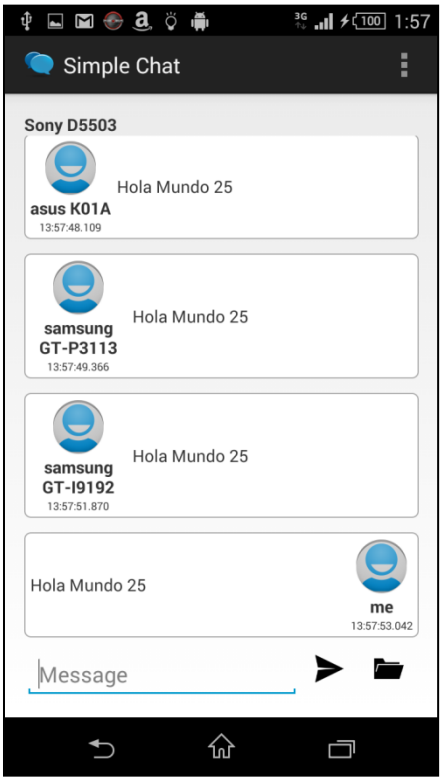


Figura 63 [Prueba1] Mensajes recibidos/enviados

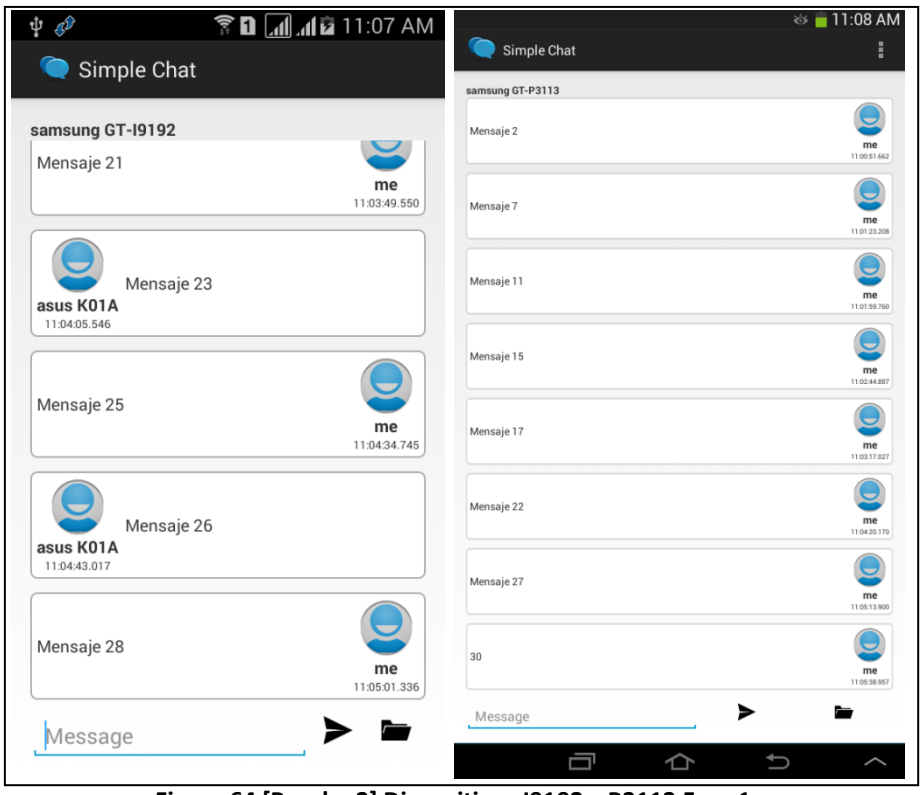


Figura 64 [Prueba 2] Dispositivos I9192 y P3113 Fase 1.

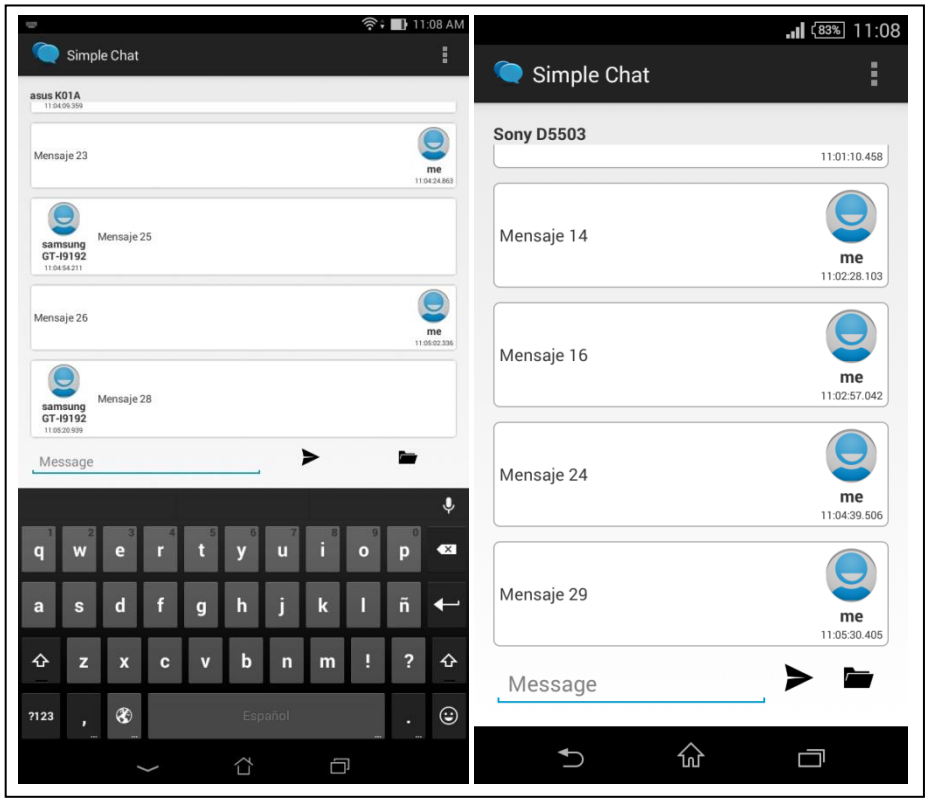


Figura 65 [Prueba 2] Dispositivos Asus y Sony Fase 1.

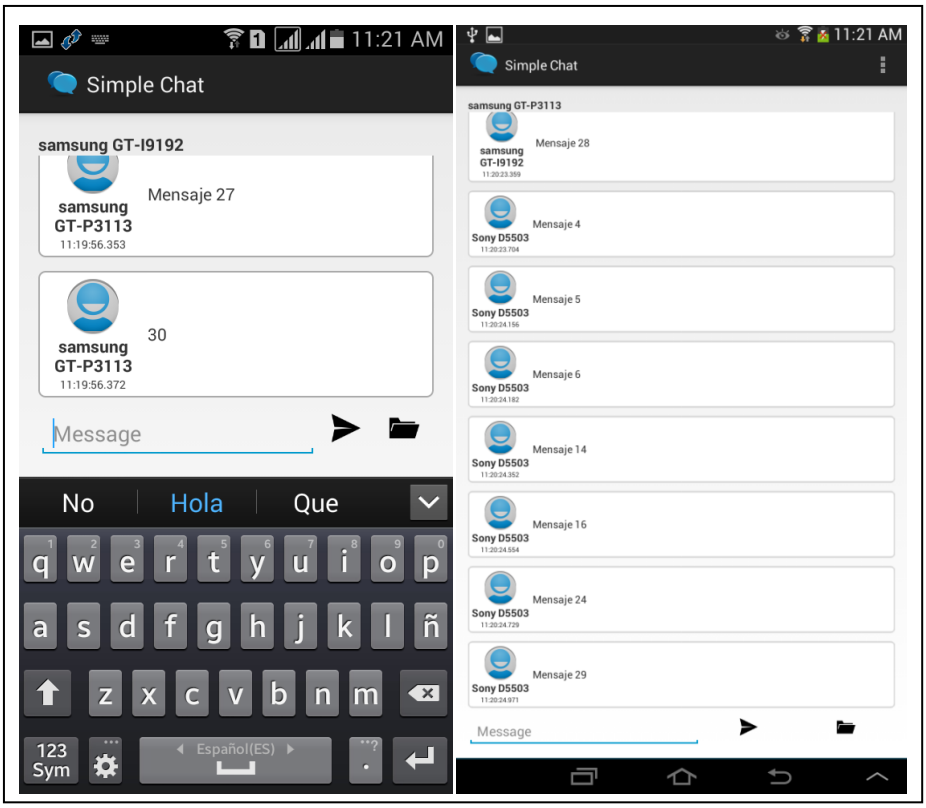


Figura 66 [Prueba 2] Dispositivos I9192 y P3113 Fase 2.

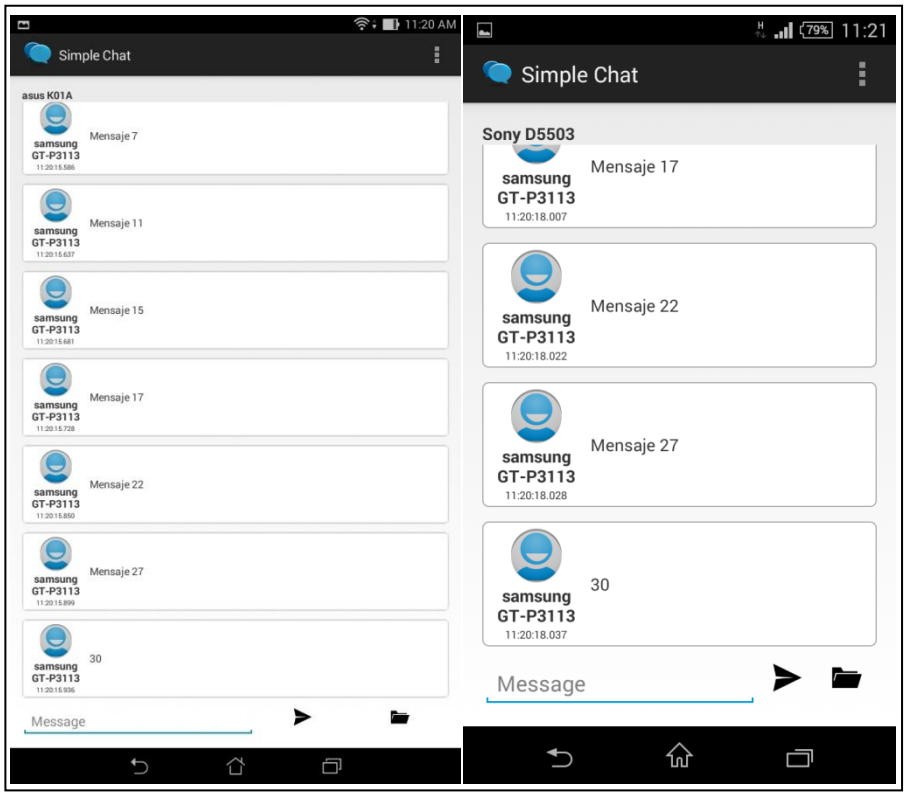


Figura 67 [Prueba 2] Dispositivos Asus y Sony Fase 2.

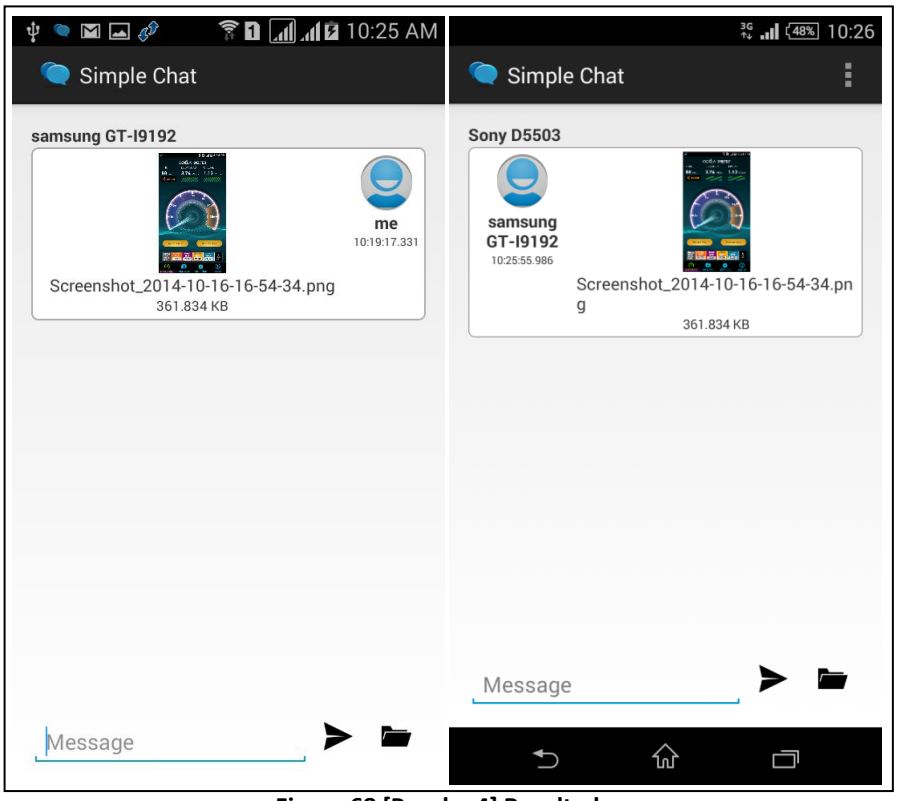


Figura 68 [Prueba 4] Resultado

Capítulo 6: Resultados y Conclusiones

Durante el desarrollo de este Trabajo Especial de Grado, se completó el desarrollo de una librería basada en tecnología push para el envío de mensajes a dispositivos móviles Android. Se utilizó la metodología de desarrollo Scrum lo que permitió una gran flexibilidad al momento de hacer algún cambio o ajuste durante cada una de las iteraciones hasta obtener el producto final.

Se desarrolló y evaluó una librería en Windows y se utilizó en varias aplicaciones. Demostrando que es posible recibir y mantener conexiones con los dispositivos clientes que se conecten al servidor. La librería servidor encoló y envió los mensajes de cada uno de los dispositivos cliente de forma exitosa. Al efecto, se hicieron pruebas con +68 mensajes de diversos tamaños y tipos, siendo satisfactorios los envíos en la totalidad de los casos de prueba.

Se desarrollaron los métodos a través de los cuales se pudo manipular los datos enviados entre los dispositivos y se proporcionaron los métodos para la comunicación entre los mismos, ya sea para enviar mensajes individualmente o a todos los dispositivos. Estos métodos pueden ser mejorados para incluir nuevas funcionalidades, como encriptación y compresión de datos.

Se diseñó e implementó un esquema de Base de Datos que contiene las tablas, vistas y procedimientos almacenados necesarios para almacenar y encolar los mensajes que envían los dispositivos. De esta forma, es posible utilizar esta información para fines estadísticos, de control y seguimiento, y para compartir con otras aplicaciones.

Se desarrolló y evaluó una librería para que pueda ser utilizada por cualquier aplicación Android que requiera mantener comunicarse vía push sin la necesidad de utilizar servidores de terceros, conexiones a internet o mantener una cuenta en Google. De esta forma, los programadores y empresas tienen la facilidad de manejar sus propios esquemas de envío de mensajes push de acuerdo a sus necesidades particulares.

Se elaboraron 3 casos de estudios utilizando las librerías desarrolladlas y se evaluó el comportamiento de las mismas realizando varias pruebas con ambientes de conexión

controlado. En la totalidad de los casos, los resultados fueron exitosos. Sin embargo, en el caso de envíos de mensajes de tamaño superior a 500K, algunos de los dispositivos de menores prestaciones tuvieron fallas, debido a inconvenientes de memoria y capacidad de procesamiento. Esto hace pensar, que debe haber algunas condiciones mínimas que deba cumplir la plataforma al momento de instalar y usar las librerías desarrolladas.

El presente Trabajo Especial de Grado se elaboró como un aporte alternativo a las opciones existentes en el mercado para el envío de mensajes push. Las librerías pueden ser utilizadas en redes públicas o privadas sin necesidad utilizar servicios externos pagos que proporcionen el servicio.

A parte de las mejoras, optimización de código y bugs que podrían surgir, algunas opciones/mecanismos que se pueden agregar al presente trabajo como posibles trabajos por venir son:

- Agregar mecanismos que permitan la unificación e integración del servicio cliente con aplicaciones del mismo fabricante. Es decir centralizar la comunicación a través de un solo servicio que sea compartido por varias aplicaciones que utilicen la misma librería. Que sea sólo un servicio de estas aplicaciones el que se mantenga activo, que se comunique con el servidor y se encargue de entregar los mensajes a todas las aplicaciones en el dispositivo.
- Incluir encriptación, compresión de imágenes/texto, manejo de archivos automáticamente.
- Buscar una forma óptima y eficiente para la recepción de archivos de tamaños considerablemente grandes (Mayores a 400KB) independientemente del dispositivo.

Referencias

1. RFC. *RFC-es.* [En línea] http://www.rfc-es.org/rfc/rfc0793-es.txt.

2. Wikipeadia. [En línea] http://es.wikipedia.org/wiki/Transmission_Control_Protocol.

3. Dashboards. *Android Developers.* [En línea] https://developer.android.com/about/dashboards/index.html?utm_source=suzunone.

4. **Burnete, Ed.** *Hello, Android - Introducing Google's Mobile Development Platform.* 2010.

5. Activities. *Android Developer.* [En línea] http://developer.android.com/guide/topics/fundamentals/activities.html.

6. Tasks and Back Stack. *Android Developer.* [En línea] http://developer.android.com/guide/components/tasks-and-back-stack.html.

7. Services. *Android Developer.* [En línea] http://developer.android.com/guide/components/services.html.

8. GCM Overview. *Android Developer.* [En línea] https://developer.android.com/google/gcm/gcm.html.

9. **Meier, Reto.** *PROFESSIONAL Android™ 4 Application Development.* s.l. : John Wiley & Sons, Inc., 2012.

10. Activity. *Android Developer.* [En línea] http://developer.android.com/reference/android/app/Activity.html.

11. Service. *Android Developer.* [En línea] http://developer.android.com/reference/android/app/Service.html.