Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Implementación de aplicaciones multiplataforma para la comunicación por voz a través de Internet

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Juan José García Giménez

Tutor: Oliver Gil, José Salvador

Curso académico 2017/2018

Resumen

El proyecto tiene como cometido poder establecer comunicación de voz entre diferentes plataformas a través de internet mediante un flujo de datos o *streaming* pudiendo entablar una conversación con razonable calidad y en tiempo real utilizando el protocolo UDP (*User Datagram Protocol*).

El proyecto utiliza el lenguaje Java para implementar tanto la aplicación de escritorio como la aplicación móvil, con una estructura basada en MVC (modelo, vista, controlador). Para la implementación de la versión móvil de la aplicación, nos hemos centrado en el sistema Android que es el sistema operativo más utilizado en dispositivos móviles.

La comunicación se puede llevar a cabo entre cualquier aplicación, bien sea de escritorio, bien sea *app* de móvil (escritorio - móvil, escritorio - escritorio, móvil - móvil).

Con el fin de optimizar la comunicación y mejorar la calidad de servicio, se abordan estrategias de resolución de problemas relacionadas con la congestión en la red, retrasos o pérdida de paquetes entre los terminales.

Palabras clave: VoIP, UDP, comunicación, Android, Java, internet, app, *streaming*, multiplataforma, audio, *jitter*.

Abstract

The purpose of this project is to carry out a voice communication between differents platforms through internet, using a data streaming encapsulated packets in UDP datagrams, allowing us, achieve a reasonable quality conversation in real time.

We use Java language to implement both desktop application and mobile application through a MVC-based structure (model, view, controller). For the mobile version, we focus on Android system, which is the most widely-used mobile operating system.

The application allows you communicate between any application, either desktop, or mobile app (desktop-mobile, desktop-desktop, mobile-mobile).

In order to optimize communication and improve the quality of service, we carry out strategies for solving problems related to network congestion, delays or packet loss between terminals.

Keywords: VoIP, UDP, comunication, Android, Java, internet, app, *streaming*, multiplatform, audio, *jitter*.

Índice

1. Introducción

1.1. Motivación

Desde sus orígenes hasta la actualidad, el uso de las redes es cada vez más utilizado. Las comunicaciones de hoy en día han avanzado a pasos agigantados permitiendo el desarrollo de nuevas tecnologías y herramientas en red que funcionan en tiempo real.

Uno de los principales retos tecnológicos al que se ha enfrentado la humanidad es que las personas puedan comunicarse de forma remota. Hoy en día, esto ya es posible sin necesidad de utilizar una línea telefónica de audio clásica.

Una de las motivaciones es que con pocos recursos podemos resolver problemas que, a priori, podrían resultar complejos. Asimismo, el reto que supone el poder comunicar diferentes máquinas a partir de este enfoque simplista, ha sido una importante contienda y motivación para el desarrollo del proyecto.

1.2. Objetivos

Este proyecto llamado *Glitchy*, pretende ilustrar y mostrar una manera de establecer comunicación de audio entre diferentes plataformas a través de internet, acercándonos aún más a un mundo que permita conectarnos independientemente del lugar donde nos situemos.

El proyecto se basa en la simplicidad tanto en el desarrollo como en la implementación del mismo. Este enfoque, intenta resolver el problema evitando el uso de librerías externas que podrían generar dependencias y hacer crecer innecesariamente el proyecto. Se evitan también entornos complejos que, en un principio, podrían facilitar la implementación pero que, siendo realista, realmente no nos es necesario su uso.

Vamos a construir la infraestructura, los protocolos y las estrategias de resolución de problemas necesarias para poder llevar a buen fin el cometido del proyecto.

1.3. Estructura de la memoria

Esta memoria se compone de 11 capítulos en los que comenzamos con un planteamiento de la situación y, poco a poco, iremos profundizando en temas más específicos y, por ende, en conceptos más técnicos.

En primer lugar introducimos al lector comentando los objetivos y la motivación para el desarrollo del proyecto. A continuación, se indican las soluciones existentes que pueden servir de base para la preparación de la aplicación y nos permiten conocer el contexto y la situación actual.

Continuamos con un análisis de la aplicación lo que nos insta a identificar los componentes necesarios que se utilizan para llevar a cabo el proyecto. Seguidamente, este análisis se complementa con una serie de conceptos básicos sobre audio y representación digital.

Profundizando en la materia, explicamos los protocolos que utilizamos para llevar a cabo la comunicación y lo complementamos describiendo las estrategias que se han adoptado para afrontar los problemas que se pueden producir en un entorno de red como internet.

Posteriormente, entramos en detalles de elaboración y arquitectura de los programas tanto en la versión de escritorio como en la versión móvil y, finalmente, concluimos con una serie de pruebas realizadas, sus resultados y las conclusiones del proyecto.

2. Contexto y situación actual

2.1. Análisis aplicaciones de comunicación sobre IP

Las comunicaciones de voz sobre IP no es un concepto nuevo y actualmente existen aplicaciones que permiten la comunicación de audio entre dos o más dispositivos. Detallamos a continuación las más destacadas.

- **Skype**: Es un software propietario con bastante trayectoria. Fue adquirido por Microsoft y su código y protocolo es cerrado y privativo. No obstante, el uso de la aplicación es gratuito [1]. Permite la comunicación por texto (chat), por voz y video sobre IP (*VoIP*).

- **Whatsapp**: Se trata de una aplicación inicialmente para dispositivos móviles creada por WhatApp Inc. fundada en 2009 y nos permite comunicarnos con otros usuarios mediante mensajes de texto. Más tarde, sobre 2015, permite realizar llamadas de voz y después de ser comprada por Facebook, permitiría videollamadas. Al igual que Skype, se trata de un software privativo, sin embargo, el protocolo utilizado es XMPP (*Extensible Messaging and Presence Protocol*) que destaca porque es abierto y extensible basado en XML [2][3].

- **Telegram Messenger**: Esta aplicación desarrollada desde 2013 se basa en un modelo de desarrollo a base de estándares abiertos. Es administrada por una organización sin ánimo de lucro y tiene un gran parecido con Whatsapp, aunque actualmente no soporta videollamadas. Una diferencia importante con los anteriores es que la parte del cliente es libre, sin embargo, el código fuente del servidor es cerrado. El protocolo utilizado está basado en un estándar abierto y se denomina **MTProto**, que puede ser utilizado por otros desarrollos no oficiales sin la obligación de pagar regalías u otras restricciones. Hasta el año 2017 no era posible realizar llamadas de voz [4].

- **Hangouts**: Este software desarrollado por Google ha evolucionado con el tiempo. Fue lanzada en 2013 y actualmente permite conversaciones entre dos o más usuarios y tiene la capacidad de realizar videollamadas de hasta 15 personas [6]. El protocolo que utilizaba (XMPP) fue sustituido por uno propietario.

- **Line**: Surgió en marzo de 2011 y, al igual que las anteriores, permite la mensajería instantánea, así como realizar llamadas bajo IP. El software desarrollado por LINE Corporation tiene su origen en Japón y es privativo [7].

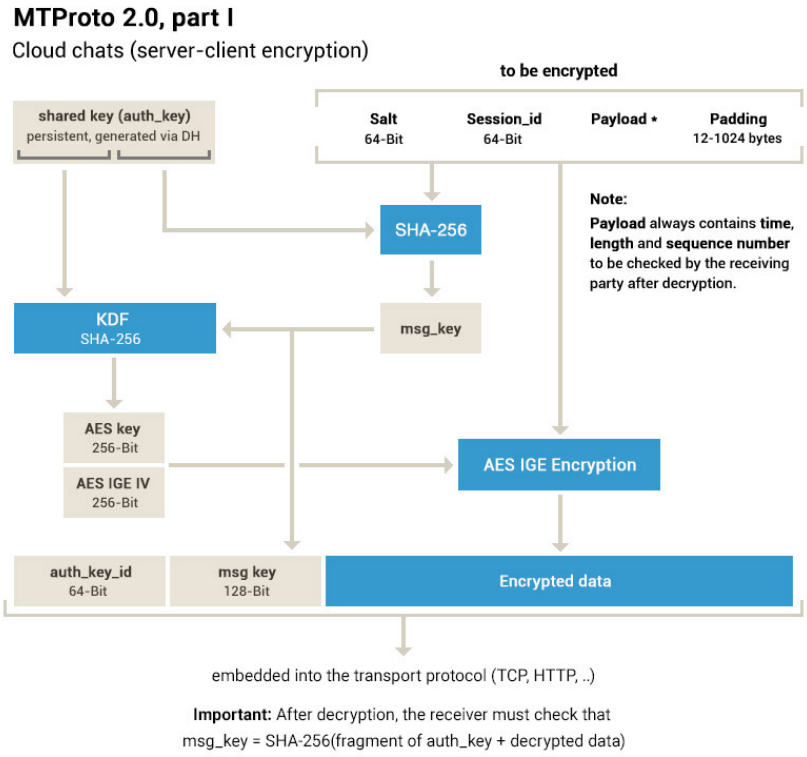


Figura 2.1.1: Protocolo de Telegram [5]

A continuación mostramos un resumen en forma de tabla para poder comparar los resultados:

Aplicación	Licencia	Protocolo	*VoIP* PC
Skype	Privativo (freeware)	Cerrrado	Sí
Whatsapp	Privativo (freeware)	XMPP	No
Telegram	GPLv2 > (código fuente del cliente)	MTProto (basado en estándar abierto)	Sí
Hangout	Privativo (freeware)	Propietario	Sí
Line	Privativo (freeware)	Desconocido [8][9]	Sí

Tabla 2.1.1: Resumen aplicaciones similares

2.2. Análisis de la situación

Como hemos comentado anteriormente, el concepto de voz sobre IP no es asunto nuevo, aunque si revisamos las fechas en las que se implementaron los programas, nos damos cuenta que su implantación en las aplicaciones más utilizadas no es tan lejana en el tiempo.

No obstante, además de los programas indicados anteriormente existe una gran variedad de aplicaciones con menor impacto orientadas al mismo cometido.

Es verdad que las comunicaciones han avanzado mucho y permiten realizar gran cantidad de operaciones. Sin embargo, debemos tener en cuenta que nuestro proyecto

intenta ilustrar la posibilidad de implementar un sistema de comunicación similar a las de las aplicaciones más utilizadas, con la peculiaridad de llevar a cabo un desarrollo desde la raíz, minimizando en gran medida el uso de librerías externas. De esta manera, evitamos posibles limitaciones, dependencias o restricciones que se nos puedan imponer.

2.3. Conclusiones

Debido que intentamos crear una aplicación, conceptualmente lo más sencilla posible y queremos evitar generar dependencias y posibles limitaciones, hemos desarrollado un sistema propio de comunicación tanto a nivel protocolario como a nivel de implementación, lo cual nos exige la necesidad de detallar unas bases suficientemente clarificadoras sobre el funcionamiento de la información digital de audio, para su posterior implantación en la aplicación.

Siendo conscientes de que existen aplicaciones con el mismo fin, nos vamos a centrar en el cómo hacerlo, aplicando unas bases que, en un futuro, pueden servir para el desarrollo de nuevas herramientas.

Dado el enfoque de este proyecto, el desarrollo podría ampliarse a nuevas metas como la transmisión de imagen, vídeo o cualquier tipo de información que necesitemos transmitir.

3. Análisis

El sistema implementado es una aplicación de comunicación por voz sobre IP que no requiere suscripción ni registro de usuarios. Se trata de un sistema completamente abierto y que nos permite comunicarnos con un extremo a través de la dirección IP y el puerto indicado. El destino deberá ser compatible con el protocolo implementado en esta aplicación.

El hecho de implementar un sistema de suscripción está fuera de los objetivos de este proyecto en el que se estudia el qué y el cómo se lleva a cabo la comunicación de datos de audio entre terminales.

El proyecto se divide en dos partes claramente diferenciadas: la aplicación de escritorio y la aplicación móvil. La versión de la aplicación de escritorio nos servirá como base a modo de librería para desarrollar la versión móvil. Más adelante veremos cómo se ha llevado a cabo el proceso.

3.1. Funcionalidades del sistema

Vamos a detallar las funcionalidades del sistema sin entrar en detalle en la implementación de las mismas.

Conexión con un destino que esté escuchando una llamada: El terminal tanto de escritorio como móvil es capaz de establecer comunicación con el destino.

Protocolo de comunicación: El sistema debe llevar a cabo un protocolo específico y ordenado con el fin de poder comunicarse con el cliente correctamente. Además que debe entender la codificación que se utiliza en el protocolo, al igual que ocurre con una llamada normal, primero se debe realizar una llamada que el destinatario podrá aceptar o rechazar.

Control de congestión: La aplicación dispone de un sistema para amortiguar los problemas de reproducción entrecortada que se puede producir por los posibles retardos que se pueden producir por la red en la entrega de los paquetes.

Control de paquetes perdidos: Existe un algoritmo de recuperación de paquetes perdidos utilizando información redundante en el flujo de comunicación.

Más adelante vamos a ver como cada una de estas funcionalidades se implementa, haciéndonos ver como un desarrollo que, a priori, puede resultar sencillo, requiere soluciones algo más elaboradas.

3.2. Definiciones y acrónimos

Esta sección define la terminología, definiciones de conceptos, acrónimos y el significado de algunos términos que se emplea en la memoria.

Terminal / extremo: Cuando hablamos de terminales en esta memoria nos referimos a cada uno de las aplicaciones que quieren comunicarse entre ellas de forma remota. Estas pueden ejecutarse en un dispositivo móvil o en una máquina de escritorio como puede ser un PC. También podemos hacer uso de la palabra extremo o cliente haciendo referencia al mismo concepto.

Servicio: La palabra puede hacer referencia a la capa que implementa la lógica de la aplicación o un servicio de Android dependiendo del contexto. En cualquier caso, se especificará a qué nos estamos refiriendo a lo largo del documento.

Usuario activo: Es la instancia que inicia la comunicación con el dispositivo remoto enviando una petición de conexión.

Usuario pasivo: Es la instancia que recibe la petición de conexión por parte del usuario activo. También podemos hacer referencia a este concepto a través del término extremo o terminal destino.

Paquete de aplicación: Cuando hablamos de paquete, estamos indicando el formato del propio paquete desarrollado en el proyecto. También puede hacer referencia a los paquetes de red (datagramas). A lo largo del documento se indicará de qué tipo estamos hablando.

Clase: Dependiendo del contexto puede significar un tipo de entidad o una representación de objeto de Java.

Clase abstracta: Se trata de una clase que se utiliza para definir subclases y, por tanto, no se puede instanciar. Puede disponer de funcionalidad y contener métodos abstractos que deberán ser implementados por las clases que hereden de la misma.

Extiende: En el contexto de implementación de clases, una clase extiende a otra si tiene una relación de herencia. Por tanto, decir que la clase *A* extiende a *B*, es igual que decir que *A* hereda de *B*.

Interfaz: Puede referirse a la pantalla de interacción que se muestra al usuario o una clase que especifica qué debe hacer pero no su implementación.

Actividad: En esta memoria se refiere normalmente al componente de Android que contiene una interfaz con la que los usuarios pueden interactuar.

3.3. Identificación de objetos del sistema

Vamos a identificar las diferentes fuentes de requisitos con el objeto de detallar los componentes que forman parte de la aplicación.

Dado que se trata de un sistema entre extremos, no existe un servidor intermediario que gestiona la comunicación.

Los actores relevantes de la aplicación son los terminales que interactúan con el sistema, cada uno de ellos con los roles que los caracterizan.

Con el fin de simplificar el análisis hemos establecido un rol de cliente y de servidor a cada usuario. No obstante, este rol depende de quién inicie la comunicación y si el dispositivo de destino está escuchando la llamada. A continuación mostramos esta información en el siguiente diagrama.

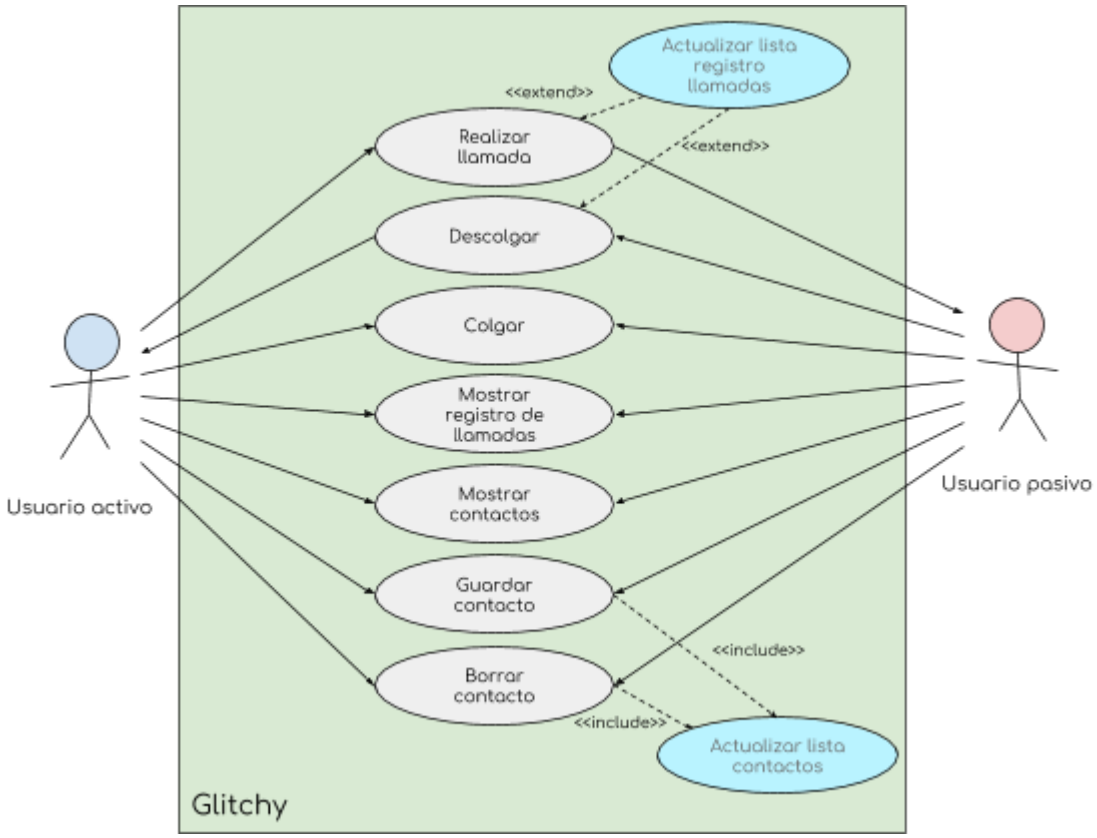


Figura 3.3.1: Casos de uso de los actores

Es importante destacar que existen dos tipos de funcionalidades. Por un lado tenemos las que solicita el usuario explícitamente. Estas acciones son las que el usuario indica a través de la interfaz de la aplicación (de escritorio o móvil).

Por otro lado, vemos las funcionalidades de sistema que no las activa explícitamente el usuario sino que el sistema es el que lo hace.

En la figura anterior vemos que tenemos dos actores, **usuario activo** y **usuario pasivo**.

El **usuario activo** es aquel que inicia la conexión con el destinatario pasivo. Se trata del usuario que desea iniciar la conversación.

El **usuario pasivo** es el que escucha y atiende la llamada del cliente.

A efectos prácticos, todo usuario del sistema con un terminal móvil es, de forma predeterminada, un **usuario pasivo**, es decir, escucha la llamada por parte de un cliente. Si este usuario pasivo inicia una llamada, pasa a ser un **usuario activo** hasta que finaliza la conversación, momento en el que volverá a pasar a ser pasivo.

En la aplicación de **escritorio**, si el usuario desea poder escuchar llamadas, deberá acceder a la opción correspondiente que consiste en pulsar un botón llamado *Escuchar* de tal manera que la aplicación permanece a la espera de llamadas.

Como podemos observar en la figura anterior, un extremo con rol de usuario activo puede llamar a un terminal destino, insertar un nuevo contacto y borrar uno existente. Si el extremo actúa como usuario pasivo puede recibir llamadas y, al igual que el usuario activo, insertar y borrar contactos, acción que implica una actualización de la lista de contactos. Hemos obviado la opción de visualizar o consultar los contactos o el registro de llamada dado que para borrar uno existente, el usuario deberá poder hacerlo.

Seguidamente, vamos a detallar los diferentes casos de uso.

3.2.1 Realizar llamada

Este caso de uso implementa la funcionalidad de llamar a un terminal destino a través de una dirección IP y un puerto. En caso de no existir la dirección o no estar el puerto abierto, la aplicación volverá al estado inicial. En el caso de la aplicación móvil, el sistema pasará automáticamente a modo de escucha de llamada si la conexión no ha tenido éxito.

Descripción	Llama a un terminal destino a través de una dirección IP y un puerto.
Actores	Usuario activo
Precondición	El terminal no deberá estar en un estado de llamada entrante, llamada saliente o llamada en curso, es decir en conversación.
Postcondición	Aparece en pantalla que se está realizando una llamada. Actualiza el registro de llamadas de la aplicación.

3.2.2 Descolgar

El usuario que es llamado puede descolgar la llamada al igual que ocurre con una llamada normal. Si el destinatario acepta, se llevará a cabo el envío de flujo de datos de audio digital. En caso contrario, se cerrará la conexión de forma ordenada.

Descripción	Descuelga una llamada.
Actores	Usuario pasivo
Precondición	El terminal deberá estar en un estado de llamada entrante.
Postcondición	Inicia los flujos de audio digital. Actualiza el registro de llamadas de la aplicación para que la llamada entrante pase a llamada aceptada.

3.2.3 Colgar

Tanto el usuario que llama como el que es llamado, tienen la posibilidad de colgar la llamada. En el primer caso, podría darse el caso de que el usuario se haya equivocado de contacto y cancele la llamada. En el segundo caso, el usuario puede no estar disponible en ese momento y decidir colgar la llamada.

Descripción	Cuelga una llamada entrante o saliente.
Actores	Usuario activo y pasivo
Precondición	El terminal deberá estar en un estado de llamada entrante, llamada saliente o en conversación.
Postcondición	Finaliza la llamada y vuelve al estado inicial.

3.2.4 Mostrar registro de llamadas

Los usuarios podrán mostrar el registro de llamadas desde la interfaz de usuario.

Descripción	Muestra el listado de registros de llamada a partir de los datos almacenados en el sistema.
Actores	Usuario activo y pasivo
Precondición	Debido al diseño de actividades en la versión móvil, el terminal no deberá estar en un estado de llamada entrante, llamada saliente o en conversación, pues la pantalla de llamada solapa a la principal. En la versión de escritorio es posible llevar a cabo este caso mientras existe una llamada en curso.
Postcondición	-

3.2.5 Mostrar contactos

Permite mostrar los contactos almacenados en los datos de la aplicación.

Descripción	Muestra los contactos de la aplicación a partir del almacén del sistema.
Actores	Usuario activo y pasivo
Precondición	Debido al diseño de actividades en la versión móvil, el terminal no deberá estar en un estado de llamada entrante, llamada saliente o en conversación, pues la pantalla de llamada solapa a la principal. En la versión de escritorio es posible llevar a cabo este caso mientras existe una llamada en curso.
Postcondición	-

3.2.6 Guardar contacto

Los usuarios podrán guardar un contacto desde la interfaz de usuario. Deberán indicar la dirección IP y el puerto que desean guardar. Esta acción implica una actualización del listado de contactos en el fichero de datos.

Descripción	Guardar un contacto a partir de los datos introducidos en el formulario.
Actores	Usuario activo y pasivo
Precondición	Debido al diseño de actividades en la versión móvil, el terminal no deberá estar en un estado de llamada entrante, llamada saliente o en conversación, pues la pantalla de llamada solapa a la principal. En la versión de escritorio es posible llevar a cabo este caso mientras existe una llamada en curso.
Postcondición	Actualiza el listado de contactos en los datos de la aplicación

3.2.7 Eliminar contacto

También es posible eliminar un contacto existente utilizando la interfaz de usuario. Al igual que en el caso anterior, esta acción implica una actualización del listado de contactos en el fichero de datos. En el caso de utilizar la aplicación móvil, el usuario deberá realizar una pulsación continua para que le aparezca la opción. Sin embargo, en la aplicación de escritorio la opción está siempre disponible.

Descripción	Elimina un contacto especificado y actualiza
Actores	Usuario activo y pasivo
Precondición	Al igual que en el caso anterior en la versión móvil, el terminal no deberá estar en un estado de llamada entrante, llamada saliente o en conversación. En la versión de escritorio es posible llevar a cabo este caso mientras existe una llamada en curso.
Postcondición	Actualiza el listado de contactos en los datos de la aplicación

3.4. Restricciones de diseño

El sistema está desarrollado con Java. Este hecho tiene más ventajas que desventajas. Una ventaja es que en el proyecto que nos ocupa hemos desarrollado una serie de clases que encajan perfectamente en la aplicación de Android. De hecho la aplicación móvil utiliza la base de la aplicación de escritorio como librería para funcionar.

Como desventaja, vemos que no es posible utilizar directamente estas clases en el sistema operativo de Apple, iOS. Sin embargo, existen herramientas que permiten el uso de Java en el desarrollo de aplicaciones móviles como Oracle Mobile Application Framework. Esta consideración es de especial importancia a la hora de llevar a cabo el desarrollo de la aplicación en iOS.

Entendemos que la implementación en iOS nos puede ofrecer un enfoque similar a la versión de Android y, por lo tanto, hemos prescindido de la misma teniendo en cuenta los fines que engloba este proyecto.

Otra característica que se debe tener en cuenta es que a pesar de que la aplicación utiliza de forma predeterminada un puerto dentro del rango de los denominados *registrados* (categoría de la IANA), éste deberá estar abierto para que la aplicación funcione. No obstante, este puerto predeterminado, puede ser modificado en las constantes de la aplicación.

Es muy importante comentar que existen limitaciones a la hora de acceder a un terminal móvil a través de la **red de datos** de internet ya que, es muy probable que el acceso se realice a través de una NAT o *Carrier-Grade* NAT [12]. NAT significa *Network Address Translation* y consiste en convertir en tiempo real las direcciones de los destinos de los paquetes para poder dirigirlos dentro de otra red. Esto nos permite intercambiar paquetes entre dos redes con direcciones incompatibles. Normalmente, estas redes son subredes privadas.

Normalmente, en una red de internet fija, disponemos de una NAT en nuestro *router* con la capacidad de abrir y cerrar los puertos que deseemos. Pero esto no ocurre en la mayoría de redes móviles. El hecho de utilizar una NAT intermedia fuera de nuestro control, supone un problema en cuanto al uso de la aplicación entre terminales móviles ya que nos encontramos con que no es posible abrir puertos, ni se puede acceder a la red desde fuera.

Dado que nuestro proyecto utiliza una dirección IP para conectar con el destino directamente, es posible que éste se encuentre detrás de una NAT y no podamos acceder.

De la información anterior, deducimos que se requiere una interfaz de red en el dispositivo que la ejecuta que puede ser inalámbrica o por cable para poder realizar la comunicación. Por tanto, será requisito indispensable que los dispositivos que se van a comunicar estén conectados a una red que les permita conectarse entre sí.

De conformidad con uno de los objetivos de la aplicación en cuanto a simplicidad de implementación, la comunicación entre los terminales no se encripta ni tampoco se comprime la información que intercambian.

3.3.1 Suposiciones

En el desarrollo de este proyecto hay ciertas características que debemos asumir como base en nuestra arquitectura.

Se asume que el sistema funcionará bajo una red que nos permita enviar paquetes con una velocidad suficiente para que el flujo de audio se envíe y reciba con fluidez. Esta velocidad podría encontrarse en un ancho de banda mínimo de 705,6 Kbps (44100 x 2 bytes x 8) por segundo. En caso contrario, los usuarios pueden sufrir cortes a pesar de disponer de un buffer que intente atenuar los retardos. Evidentemente, si el buffer se vacía, no hay datos que reproducir hasta que se vuelva a llenar.

Existen estrategias para reducir en gran medida el consumo de ancho cuando se producen silencios, pero somos conscientes que el hecho de que se disponga de un ancho de banda suficiente, no quiere decir que no se pueda perder paquetes.

Se supone que se utilizan unos terminales con suficiente potencia de cálculo como para poder interpretar los paquetes y ejecutar con fluidez la aplicación. En caso contrario, los usuarios pueden sufrir cortes provocados por el rendimiento del dispositivo.

4. Conceptos generales

4.1. El sonido digital

A groso modo, el audio digital se puede identificar como una secuencia de bytes que serán interpretados según los parámetros de muestras/segundo, resolución por muestra y canales de audio.

4.1.1. Introducción

Desde el punto de vista físico, el sonido son ondas que se transmiten como variaciones de presión y se propagan por el aire. Esta presión sonora, se recibe a través del micrófono que convierte ondas acústicas a señales de audio cuyo voltaje es proporcional a la intensidad de la onda acústica que le llega [11].

La señal analógica del micrófono, se digitaliza, mediante un muestreo, que consiste en tomar valores o muestras cada cierto tiempo. La frecuencia con que se obtienen los valores es lo que se denomina *sample rate* o frecuencia de muestreo.

Una vez obtenidas las muestras, se debe obtener los valores de cada una de ellas. Este procedimiento se lleva a cabo mediante un proceso de cuantización que permite representar cada muestra con un número de bits concreto, por tanto, a más resolución mayor similitud con la onda original.

4.1.2. Algunos términos relacionados con la transmisión digital multimedia

- Sample rate: Es la frecuencia de muestreo expresado en Hz.
- Channels: Es el número de canales del sonido (estéreo, monoaural, 5.1, etc.).
- Frame size: Es el tamaño del paquete que contiene la información de audio.
- Frame rate: Es la cantidad de información que se transmite por segundo (frames / s).

4.1.3. Resolución de muestra

Los bits por muestra nos permite saber qué resolución se utiliza para cada muestra de la forma de onda, o cuantos niveles de amplitud podemos definir por muestra. Estos niveles definen la resolución final de la onda. Las más comunes son de 16 bits, pero también existen de 8 y 32 bits.

Por ejemplo, si tenemos una secuencia en la que cada byte representa la amplitud de la muestra en la forma de onda, podemos deducir que se trata de un sonido de 8 bits por muestra, pues cada byte puede representar 8 bits, es decir 256 valores de amplitud.

Sin embargo si sabemos que el audio que estamos tratando tiene una resolución de 16 bits esto quiere decir que para representar cada muestra, necesitaremos dos bytes (8 + 8), es decir 65536 valores.

Dado que una forma de onda tiene tanto valores positivos como negativos, los valores anteriores de 256 para 8 bits así como los 65536 para 16 bits contemplan valores positivos y negativos. El tipo byte en Java se representa con signo y por tanto el mínimo y máximo valor es -128 y 127 respectivamente. Esto lo debemos tener en cuenta a la hora de gestionar los números de secuencia.

Por tanto con 256 valores podremos definir una muestra entre -128 y + 127. En el caso de 16 bits -32768 y +32767. Los rangos indicados representan los valores máximos posibles en la onda.

A continuación vemos una ilustración en la que se compara la onda original con las diferentes versiones digitales.

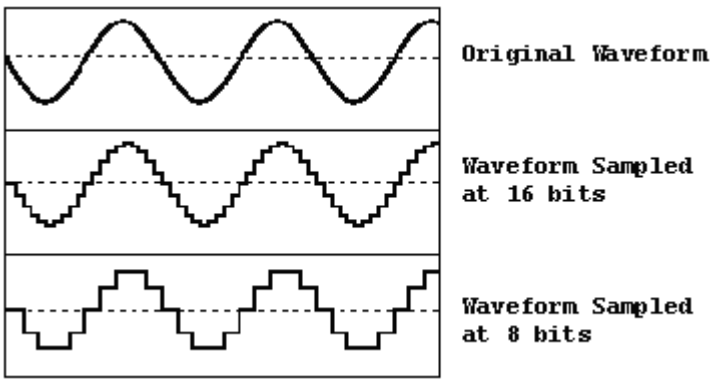


Figura 4.1.1.1: Muestreo de una forma de onda [10]

Evidentemente a mayor resolución, mayor cantidad de bytes a enviar al hacer *streaming*.

4.1.4. Frecuencia de muestreo

La frecuencia de muestreo o *sample rate* no es más que el número de muestras que se procesan por segundo. Se representa en hercios que significa ciclos por segundo.

Esta frecuencia afecta a la calidad final del audio y, normalmente varía entre 11025 Hz y 96000 Hz (BlueRay). No obstante, en aplicaciones de transmisión de voz sobre IP suele ser de 8000 Hz.

Dado que la frecuencia determina la cantidad de muestras que se han de procesar por segundo, esta característica afecta a la hora de transmitir audio a través de la red.

4.1.5. Número de canales

El número de canales es la cantidad de líneas de audio que se escuchan a la vez. En una señal estéreo escuchamos dos formas de onda distintas en los canales izquierdo y derecho. Una señal 5.1 utiliza cinco canales, dos frontales (izquierdo y derecho), uno central, dos canales de sonido envolvente y 1 canal de baja frecuencia (*subwoofer*).

Cada línea lleva un sonido y una forma de onda como las que hemos comentado anteriormente.

Para transmitir un audio estéreo se necesitará el doble de flujo de bytes que si la señal es monoaural.

5. Diseño y componentes

5.1. Dispositivos

El sistema no requiere un sistema complejo para poder funcionar. Dado que el proyecto se ha implementado en terminales móviles hoy en día obsoletos, cualquier dispositivo adquirido en la actualidad, dispondrá de capacidad suficiente como para permitir llevar a cabo el objetivo de forma fluida. A continuación, vamos a indicar las necesidades mínimas para su correcto funcionamiento.

Se requiere una interfaz de red en el dispositivo que la ejecuta que puede ser inalámbrica o por cable para poder realizar la comunicación. Por tanto, será requisito indispensable que los dispositivos que se van a comunicar estén conectados a una red que les permita conectarse entre sí.

Como hemos comentado en el apartado de "3.4 Restricciones de diseño", de forma predeterminada, la aplicación utiliza el puerto 20001 para establecer la comunicación. No obstante, es posible modificar este parámetro desde la interfaz del usuario.

5.2. Entorno de desarrollo

Para la implementación de las aplicaciones se ha optado por utilizar el lenguaje de programación Java por su versatilidad y cuyo desarrollo estaba orientado a la capacidad de ejecutarse en diferentes plataformas (multiplataforma).

Figura 5.2.1: Lenguaje de programación Java

El entorno de desarrollo utilizado para la implementación tanto de la aplicación de escritorio como la del dispositivo móvil (Android) es Eclipse, más específicamente sus versiones Mars y Neon. Se trata de un programa de código abierto ampliamente utilizado en entornos corporativos por su facilidad de ampliar la funcionalidad mediante módulos (plugins) y su gran robustez. Además, este entorno está pensado principalmente para el lenguaje Java.

Figura 5.2.2: Entorno de programación Eclipse

Como hemos comentado anteriormente, hemos intentado evitar al máximo el uso de librerías específicas para desarrollar la aplicación, tan sólo son necesarias las librerías predeterminadas de un proyecto Java estándar (Java SE-1.8) y además la librería *org.json* dado que se hace uso del formato JSON (JavaScript Object Notation) para gestionar los datos de aplicación.

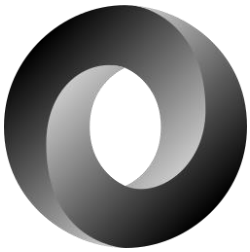


Figura 5.2.3: Formato de texto ligero JSON (JavaScript Object Notation)

No hemos tenido la necesidad de utilizar un gestor de proyectos tipo Maven, ni un sistema de versionado como SVN (Subversion). Cabe destacar, que una de las principales motivaciones para el desarrollo de este proyecto es la simplicidad tanto en el desarrollo como en la implementación del mismo. No obstante, es recomendable su uso.

En cuanto al almacenamiento de datos, la información se guardará de forma estructura en un fichero de texto en formato JSON. Aunque es recomendable el uso de base de datos y sistemas basados en SQL (Structured Query Language) por su escalabilidad, no hemos tenido la necesidad de utilizarlo dado el objeto del proyecto. No obstante, la aplicación está estructurada de tal manera que la implementación de un sistema que acceda a base de datos no supondría un cambio necesariamente complejo, pues se dispone de una capa de acceso a datos DAO en la que implementar estas modificaciones sin afectar al resto de la aplicación.

6. Protocolos y estructura de los paquetes

En cuanto a nivel de transporte que supone la capa 4 del modelo OSI (Open System Interconnection) [14], la aplicación utiliza el protocolo UDP para el envío de datos entre los terminales. Este protocolo proporciona una manera de que las aplicaciones puedan enviar datagramas IP (nivel 3) encapsulados, pero hay que tener en cuenta que no está orientado a la conexión, es decir, los terminales envían paquetes sin asegurarse si el destino está disponible.

Además, una vez que la máquina destino reciba el paquete, ésta no tiene por qué confirmar nada al origen. Por tanto, funciona en un modo *Best-Effort* ya que no hay garantías de entrega.

Hemos optado por este protocolo por tres razones que detallamos a continuación:

1. **Simplicidad**: La simplicidad de los paquetes UDP reduce la cantidad de información necesaria para llevar a cabo la comunicación. UDP no solicita respuestas como hace TCP.
2. **Velocidad**: En aplicaciones en tiempo real, la confiabilidad no tiene por qué ser crítica. Buscamos velocidad en la comunicación. El hecho de perder algunos datos de audio, puede ser aceptable en nuestro caso. Si un paquete de audio no llega en el momento relativamente razonable, no es necesario volverlo a enviar dada la necesidad de instantaneidad de la aplicación. Para estos casos se han implementado algoritmos que intentan recuperar esa información en la medida de lo posible enviando información redundante en la comunicación.
3. **Control**: Queremos desarrollar un sistema que nos permita poder gestionar la comunicación a un nivel más bajo y poder controlar el flujo de datos con el objeto de tener más juego a la hora de implementar algoritmos de comunicación.

La gran ventaja de UDP es que provoca poca carga adicional en la red ya que es sencillo y emplea cabeceras muy simples.

Las garantías para la comunicación se implementan a nivel de aplicación.

6.1. Estructura de los paquetes de aplicación

Hemos establecido un formato estructurado en los datos de los paquetes que enviamos y recibimos para poder llevar a cabo maniobras de optimización y resolución de los problemas relacionados con la congestión y pérdida de datagramas.

Anteriormente hemos comentado que a nivel de transporte se utiliza UDP para enviar y recibir los datos. Los paquetes de aplicación que estamos comentando en este apartado formarán parte de los datos del datagrama UDP.

A continuación detallamos la estructura del paquete:

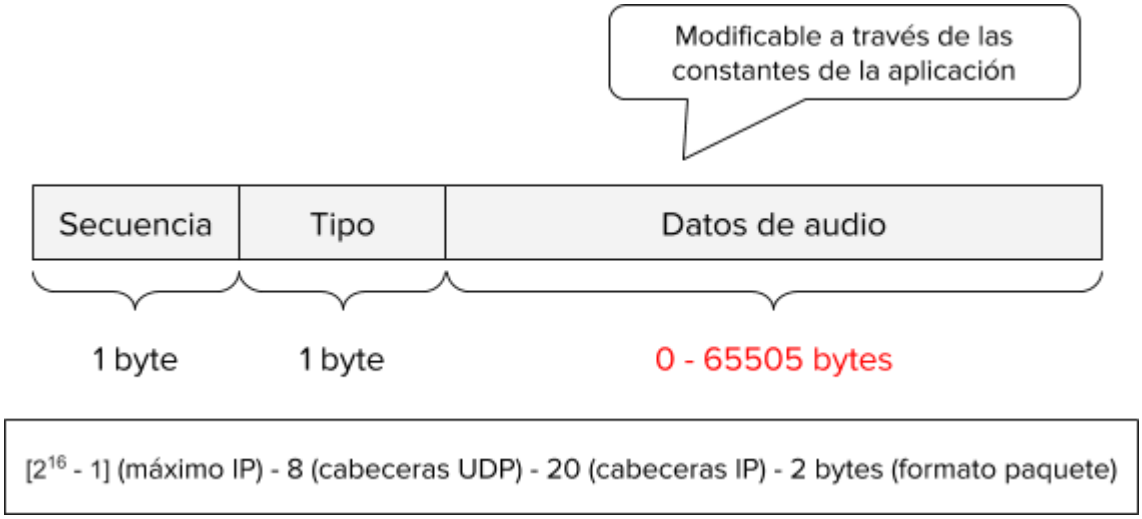


Figura 6.1.1: Formato de los paquetes de aplicación

Como podemos observar, el paquete tiene una estructura bastante sencilla: un número de secuencia, un tipo y los datos de audio. Vamos a describir cada uno de los campos:

- **Número de secuencia**: Este byte almacena el número de secuencia del paquete. Este valor es rotativo y cubre desde 0 hasta 255. Nos permite identificar los paquetes que van antes o después dentro del buffer donde se insertan antes de ser reproducidos y que comentaremos con más detalle más adelante. El número máximo de secuencia es modificable desde las contantes de la aplicación (*Constantes.Configuracion.maxSecuencia*).
- **Tipo**: Actualmente hay definidos 4 tipos que definen el tipo o los datos lleva el paquete: datos, silencio, paridad y control.
- **Audio**: No es más que la información muestreada de audio digital. Su valor depende del campo **tipo**.
  El valor máximo razonable que hemos obtenido experimentalmente sería de unos 22 KB. Es posible indicar valores más grandes pero, dado que requiere más tiempo para almacenar la información en el paquete, puede suponer un retardo en la conversación. Además, si se pierde un fragmento, se pierde todo el paquete y por tanto mucha información útil. No obstante, este valor es modificable desde las constantes de la aplicación.

6.2. Tipos de paquetes

Como se ha indicado anteriormente, existen varios tipos de paquetes, concretamente cuatro. A continuación vamos a detallar el significado de cada uno de ellos.

6.2.1. Tipo 1 - control

Un paquete de tipo control, permite llevar a cabo el protocolo de comunicación del que hablaremos más adelante. Son los tipos de mensaje que se envían al realizar una llamada, así como para comprobar si una llamada está en curso.

6.2.2. Tipo 2 - datos

Este tipo de paquete es bastante intuitivo y no tiene demasiada complejidad ya que describe un paquete que contiene datos de audio.
Cuando enviamos un paquete de este tipo, estamos indicando que el campo datos lleva información de audio digital.

6.2.3. Tipo 3 - silencio

Los paquetes de tipo silencio, indican que se trata de un paquete sin información de audio relevante y no contienen información de audio digital. Utilizamos la palabra relevante para referirnos a que la aplicación comprueba si hay información útil en el paquete antes de establecer el tipo a 'silencio'.
Este tipo de paquetes no llevan datos en el campo de **datos**, dado que estamos indicando que se debe introducir un silencio en el destino. Por tanto, nos permite ahorrar ancho de banda, pues es el destinatario el encargado de reproducir este silencio cuya duración equivaldría a la de un paquete.

6.2.4. Tipo 4 - paridad

Este paquete no es reproducible y se trata de información redundante con el fin de reconstruir un paquete de datos que sea erróneo dentro de un grupo de **n** consecutivos.

Más adelante, veremos con más detalle la manera de calcularlo y cada cuanto se hace.

6.3. Encapsulado de información

Ahora que sabemos la manera de representar audio digital en un formato de datos que podemos reproducir y la estructura de los paquetes, podemos detallar la manera de enviar esta información a través de la red.

6.3.1. Lectura de audio

En primer lugar vamos a leer la información que se «escucha» desde el dispositivo para almacenarla en memoria.

La forma de leer la información que se escucha por el micrófono del dispositivo es muy dependiente de la plataforma. Es por ello, que implementamos una clase abstracta de la que heredarán las clases que quieran enviar audio.

Mediante las librerías de cada plataforma inicializamos el micrófono y leeremos la información en forma de *array* de bytes. Con el fin de simplificar la implementación, obtenemos el micrófono que exista en la máquina de forma predeterminada. La aplicación no muestra la lista de micrófonos disponibles para seleccionar uno.

En Android, el formato de audio más extendido y que funciona en la mayoría de dispositivos es el que trabaja a 44100 Hz con una resolución de muestra de 16 bits. Este formato nos da más garantías de que el dispositivo que va a hacer uso de la aplicación es compatible. Con el fin de evitar transmitir más información de la necesaria, hemos decidido que el audio sea monoaural, es decir un canal en lugar de dos, en cuyo caso se trataría de una señal estéreo, lo cual es innecesario para llevar a cabo una conversación de voz.

Dado que estamos trabajando a 16 bits, se utilizarán 2 bytes para representar el valor de una muestra. En este punto, debemos tener en cuenta la manera en que se almacena la información, pues puede ser en orden "natural" (*big-endian*) u orden invertido (*little-endian*).

En el primer caso (*big-endian*) los bytes se almacenan de manera que los de mayor peso se almacenan primero, antes que los de menor peso. En *little-endian* es al revés, los de menor peso se almacena en posiciones más bajas de memoria.

Veamos una ilustración para ver la situación más claramente.

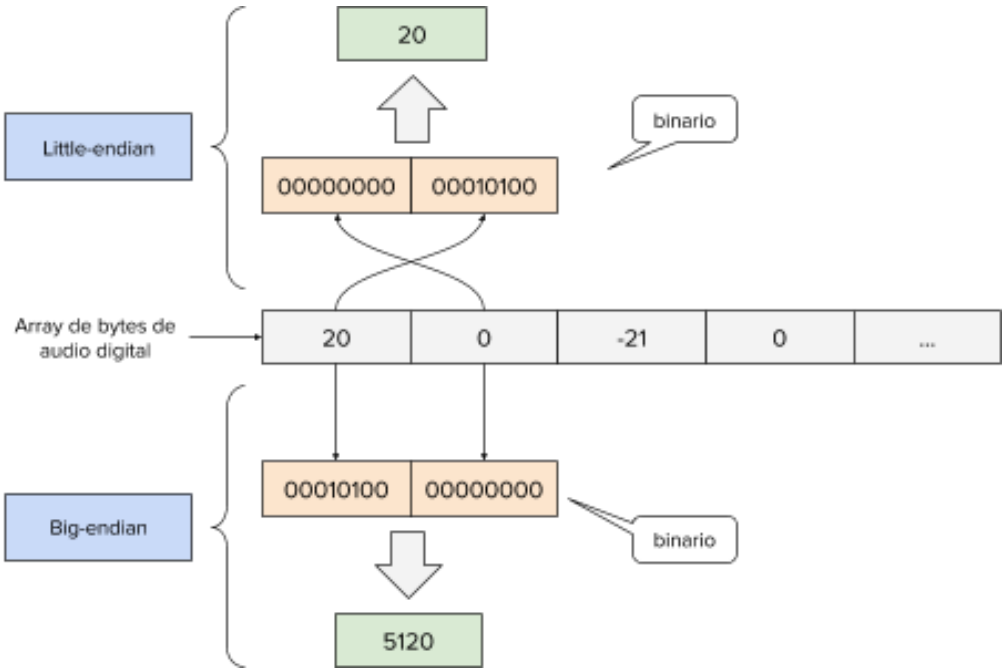


Figura 6.3.1.1: Diferencia big-endian y little-endian

Tanto la aplicación de escritorio como la de Android utilizan por defecto, el orden ***little-endian*** a la hora de insertar los niveles en el *array* de bytes. Sabiendo esto, deberemos decodificar la información para reproducir el audio correctamente.

Una vez configurado el micrófono para que lea la información en el formato indicado, encapsulamos la información en un paquete de aplicación. Remarcamos "paquete **de aplicación**" para no confundirnos con el datagrama UDP.

Existe una clase *Paquete* que nos permite realizar la encapsulación. Este paquete dispone de los atributos necesarios para representar el paquete de aplicación comentado anteriormente.

En la siguiente ilustración, podemos apreciar visualmente como se encapsula la información de audio digital sobre el paquete de aplicación y seguidamente, esta pasa a formar parte del datagrama UDP.

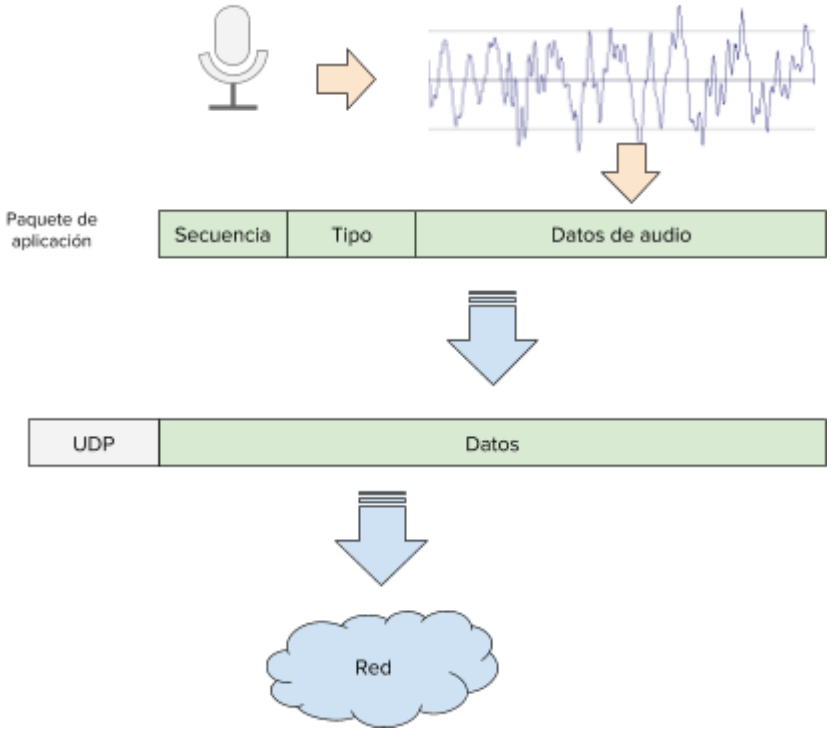


Figura 6.3.1.2: Encapsulamiento de los datos

6.3.2. Reproducción de audio

Al igual que ocurre cuando leemos la información de audio, al reproducir audio debemos tener en cuenta el formato en el que va a enviar la información al reproductor.

Por tanto, el formato de audio del emisor deberá coincidir con el del receptor en cuanto a la interpretación de los datos. En caso contrario, podemos tener problemas de cortes y velocidad de reproducción.

Cuando el datagrama llega a su destino será desencapsulado para poder acceder a los datos de audio y enviárselos a la tarjeta de sonido para que los transforme a información analógica y esa señal llegue al altavoz. Los paquetes de tipo paridad y control no son reproducibles. El paquete de tipo silencio se interpretará para generar un silencio en el receptor.

Al igual que en el apartado anterior, vamos a mostrar una figura ilustrando, de forma general, el proceso.

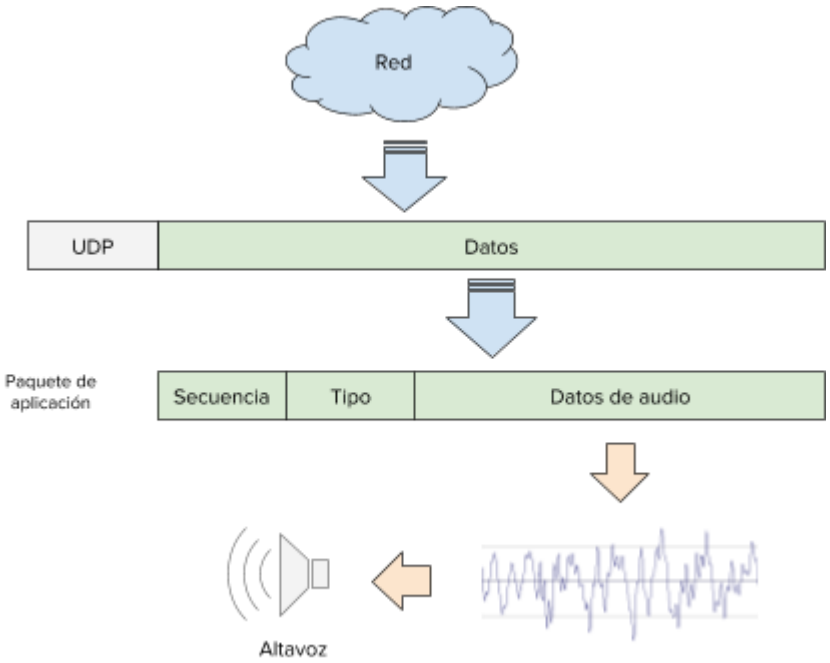


Figura 6.3.2.1: Desencapsulamiento de los datos

6.4. Protocolo de comunicación para realizar una llamada

La aplicación utiliza un protocolo propio para realizar la comunicación y llevar a cabo el sistema de llamadas. Estos mensajes se envían encapsulados en el paquete de aplicación y son de tipo *control* (tipo 1). Veamos un ejemplo general de cómo se realizaría una llamada sin entrar en detalles de implementación.

Teniendo en cuenta que los terminales deben de estar a la escucha de una llamada entrante, supongamos que ***A*** quiere realizar una llamada a ***B*** y entablar una conversación.

1. *A* lanza el método ***conectar()*** que envía el mensaje "conectar" a *B*.
2. *A* queda a la espera de la respuesta de *B*.
3. *B*, que permanecía a la escucha, recibe el mensaje de "conectar" y envía por la misma conexión el **mensaje de respuesta** de conexión "conectado".
4. *B* pasa al estado de **cliente conectado** y lanza el método ***recibirLlamada()*** que realiza los pasos necesarios para que *B* escuche el mensaje de **llamada saliente** (que envía *A*), pues supone que *A* lanzará ese mensaje después de conectar.
5. *A*, que permanecía a la espera del **mensaje de respuesta** de conexión, pasa al estado "conectado" y lanza el método ***llamar()*** que envía un mensaje de "llamadaSaliente" a *B*.
6. *A* permanece a la espera de la respuesta de B (colgar o descolgar).
7. *B* recibe el mensaje de "llamadaSaliente" y pasa al estado de **llamada entrante**. En esta situación, *B* tiene la opción de colgar o descolgar.

8. *B* descuelga y envía el mensaje "descolgar" a *A*. En ese momento, *B* lanza los hilos de audio (uno de entrada y otro de salida).
9. *A* recibe la respuesta "descolgar" de *B* y lanza los hilos de audio.

En la siguiente ilustración, vemos de una manera visual en un diagrama de secuencia el funcionamiento del protocolo de llamada. Como podemos observar, es muy sencillo a la vez que intuitivo.

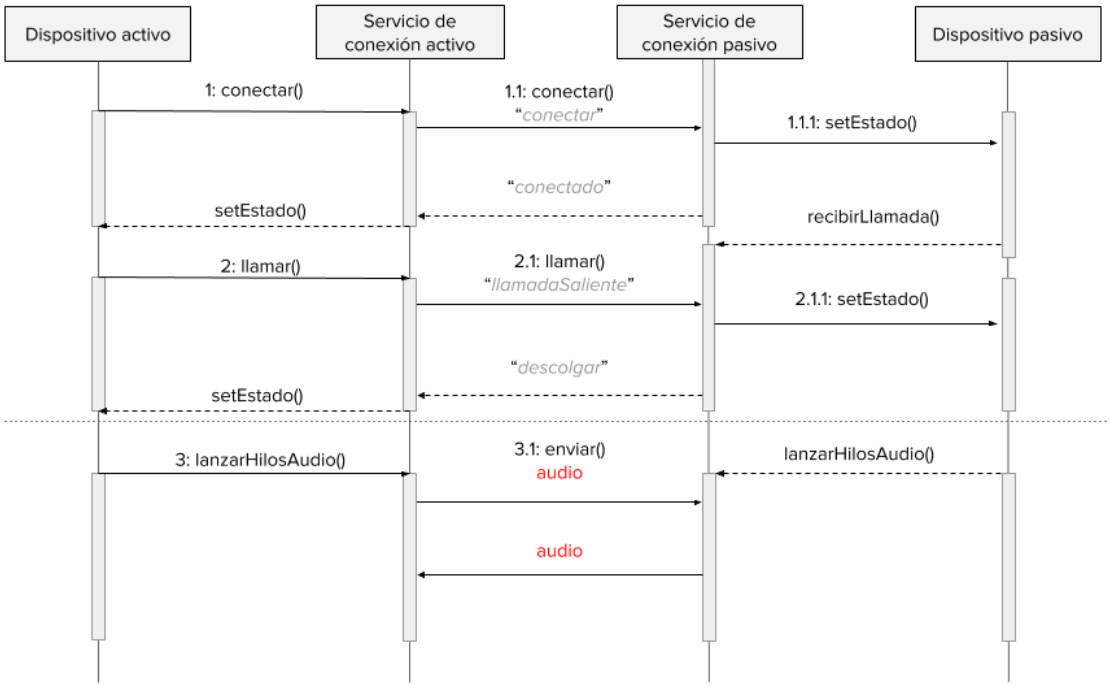


Figura 6.4.1: Protocolo básico de comunicación (descolgar llamada)

En la figura observamos que hay diferentes entidades que realizan diferentes tareas, pues la aplicación utiliza diferentes clases y procedimientos en diferentes hilos para llevar a cabo el proceso de modo que resulte una experiencia no bloqueante.

6.5. Control del estado de una llamada en curso

En esta situación, *A* y *B* están comunicados mediante una serie de paquetes de aplicación que contienen el flujo de datos con el audio muestreado que se envía a través del micrófono de cada terminal.

Como hemos comentado, existe un flujo de salida y otro de entrada que trabajan sobre la misma conexión pero en dos hilos distintos. Esta comunicación se realiza sobre el puerto 20001, de forma predeterminada.

Cuando se acepta una llamada entrante, se lanza un procedimiento para controlar si la llamada sigue vigente y está disponible. Este procedimiento consiste en realizar

comprobaciones regulares del estado de la llamada. Es decir, cada 5 segundos, los terminales envían información por el canal de comunicación para comprobar si el otro extremo sigue en pie.

A continuación, mostramos una figura que muestra los objetos y los hilos que se crean cuando se lleva a cabo una conversación en curso.

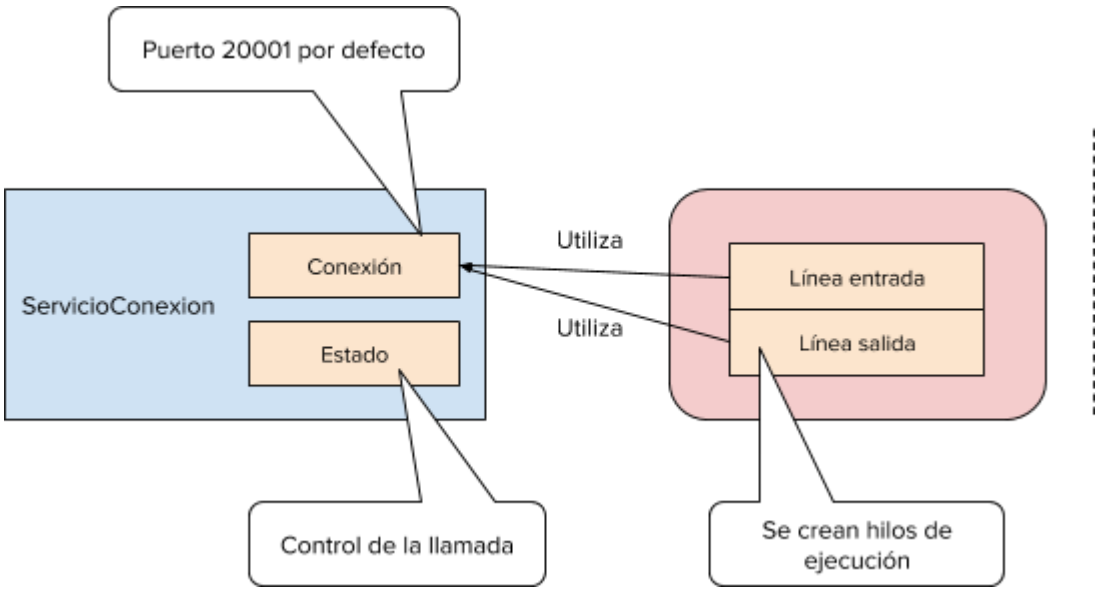


Figura 6.5.1: Conexiones e hilos de ejecución [16]

Tanto en el *log* de la aplicación de escritorio como en la sección de depuración de la aplicación móvil, podemos visualizar la recepción de los mensajes de estado.

Si la aplicación no recibe los mensajes correctos o llegan fuera de un tiempo límite de 15 segundos, se interpretará que no hay enlace entre los extremos y se desconectará el vínculo para volver al estado inicial de la aplicación.

7. Estrategias de resolución problemas de red

En comunicaciones a través de la red es posible que existan retardos o que se pierdan paquetes. Es por ello, que se han implementado varias estrategias que resuelven algunos de estos aspectos. Veremos, a continuación, cómo hemos afrontado esta problemática.

7.1. Buffer de entrada

El buffer de entrada o contención, nos permite atenuar los posibles **retrasos** que se produzcan en la red, almacenando los paquetes en un espacio de memoria temporal desde donde posteriormente serán tratados.

Este buffer también nos permite **organizar paquetes** que hayan sido recibidos de forma desordenada. Supongamos, por ejemplo, que se emiten los paquetes 4 y 5 y debido a las colas en los dispositivos de red, el paquete con número de secuencia 5 llega antes que el 4. En esta situación, si conservamos el orden de llegada, el audio resultante no sería el correcto.

El buffer está implementado con un mapa ordenado *TreeMap* con lo cual en el momento en el que se inserte un nuevo paquete en el mismo, ocuparía el lugar correspondiente.

Vamos a ilustrar el caso anterior con un ejemplo que mostramos en la figura siguiente:

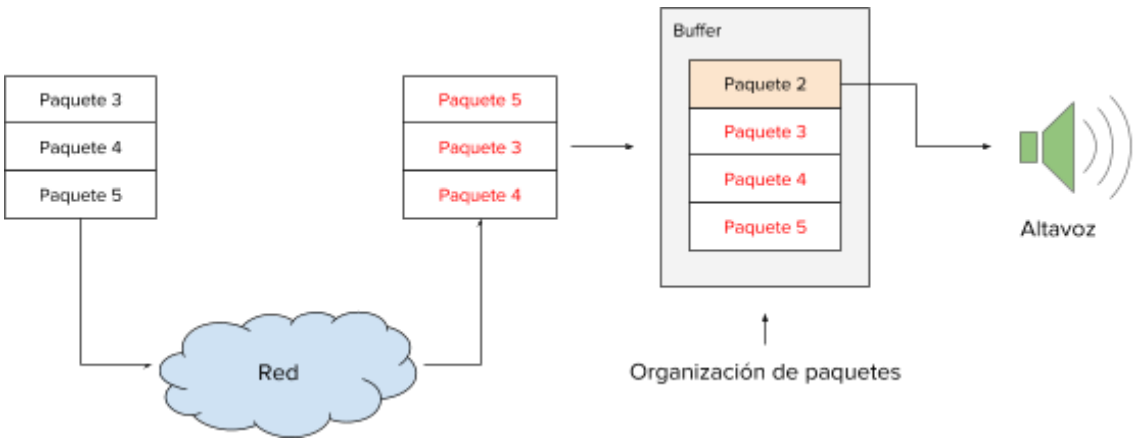


Figura 7.1.1: Reorganización de los paquetes recibidos

Evidentemente, si el paquete que nos llega, contiene un número de secuencia que no está dentro de un rango razonable de reproducción, se descartará. Por ejemplo, si recibimos el paquete 2 después de haberse **reproducido** el 3, entendemos que ha llegado demasiado tarde y se descarta.

El reproductor de audio, que se ejecuta en un hilo distinto, irá cogiendo paquetes del buffer obteniendo siempre el primero de la lista de tal manera que cuando vaya a reproducirse, se marca como **no disponible** con el fin de que no se tenga en cuenta para realizar tareas de comprobación de errores.

Una vez el paquete se reproduce, se elimina del buffer quedando el siguiente elemento, como el primero de la lista y así sucesivamente.

Debido a lo anterior, el buffer puede sufrir fluctuaciones en cuanto al número de paquetes disponibles en el mismo. Si el emisor no envía paquetes, este buffer se irá vaciando a medida que se reproducen los paquetes hasta que se llegue a cero elementos. Es por ello que es de vital importancia que la frecuencia de lectura y reproducción coincida entre los terminales. No es lo mismo reproducir audio a 22050 Hz que a 44100 Hz dado que el tiempo que requiere el dispositivo para reproducirlo varía según el caso.

El número de paquetes que debe gestionar el buffer de contención es modificable en las constantes de la aplicación (*Constantes.Configuracion.bufferContencion*).

7.2. Buffer de salida

Es importante tener en cuenta que la aplicación utiliza un buffer de salida con el fin de disponer de la información suficiente para poder calcular la información de **paridad** necesaria con la que poder recuperar paquetes perdidos.

Se trata de un buffer temporal dado que en cuanto se genera un paquete de audio de salida, se envía directamente a la red, pero esto no significa que el paquete se borre sino que es almacenado en este buffer para posteriormente poder calcular la paridad.

En cuanto se calcula la XOR, el buffer se vacía para volver a llenarlo con los paquetes necesarios para calcular la siguiente XOR.

A continuación, ilustramos el funcionamiento cuando se ha calculado la XOR y el buffer ya ha cumplido su cometido.

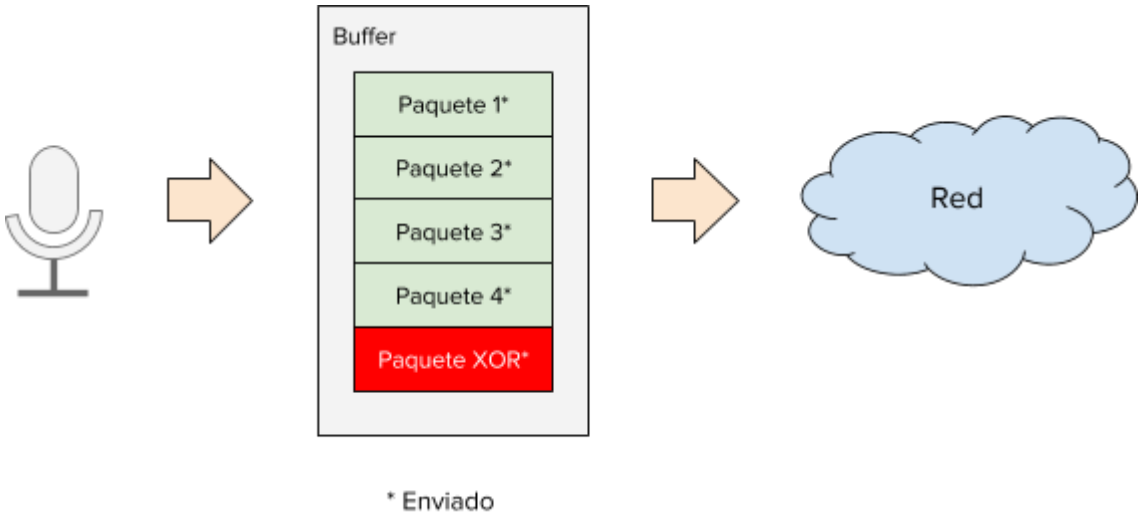


Figura 7.2.1: Buffer de salida

Todos los paquetes mostrados en la ilustración, se envían lo antes posible sin esperar a que se llene el buffer. Sin embargo, en el buffer de entrada, la aplicación se espera a que disponga de los paquetes mínimos y entonces empieza a reproducirlos ordenadamente.

Para marcar si un paquete ya se ha enviado para no ser procesado en el algoritmo de recuperación, se utiliza el campo **disponible** de la clase *Paquete*.

Con lo comentado anteriormente, podemos deducir que el tamaño máximo de este buffer de salida depende de la constante que especifica cada cuántos paquetes hay que calcular el de paridad (*Constante.Configuracion.numPaquetesParidad*). Por ejemplo, si la constante marca 6, se guardarán 5 paquetes y el número 6, será la XOR de los 5 anteriores.

7.3. Ahorro de tráfico mediante silencios

Como hemos comentado anteriormente, existen paquetes de tipo silencio (tipo 3) que nos permite ahorrar tráfico que se envía a la red y por tanto ancho de banda, pues, únicamente enviamos el número de secuencia y el tipo de paquete (2 bytes), con el fin de que el destinatario interprete esta información como silencio a la hora de reproducirlo.

Para etiquetar un paquete como silencio, hemos implementado un algoritmo que realiza un barrido rápido sobre los datos de audio del paquete. Cuando no se detecta una amplitud de onda suficiente como para transmitirla a través de la red, la aplicación interpreta un silencio en el emisor y en lugar de enviar el audio digital con los datos, envía un paquete de este tipo. De esta manera no es necesario enviar 4096-22050 bytes, únicamente enviamos 2 bytes, la secuencia y el tipo de paquete que, en este caso, es de silencio.

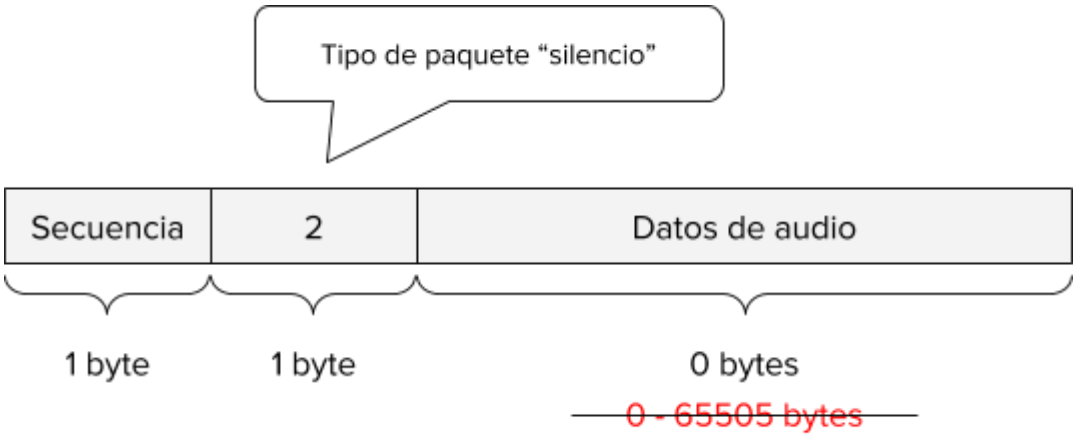


Figura 7.3.1: Paquetes de tipo “silencio”

Esto nos permite que, en los momentos en que los extremos no estén utilizando datos relevantes (sonido de fondo, silencios largos, etc.), no se envíe esta información ya que, a nivel práctico, no aporta nada.

Existe una variable que nos permite ajustar el umbral mínimo que deberá superar la forma de onda del paquete para interpretar la información como relevante. Esta variable es modificable en las constantes de la aplicación (*Constante.Configuracion.umbralMinimo*) y su valor se debe calcular en base a niveles de resolución de muestra de 16 bits.

7.4. Corrección de errores mediante XOR

Como se ha comentado anteriormente, esta estrategia permite reconstruir paquetes perdidos; uno por cada grupo de **n** paquetes. Es decir, cada **n** paquetes, uno es de paridad que permite corregir uno de ellos. Este paquete es calculado con la **XOR** de los **n-1** anteriores.

Vamos a ilustrar la manera de creación de este tipo de paquetes:

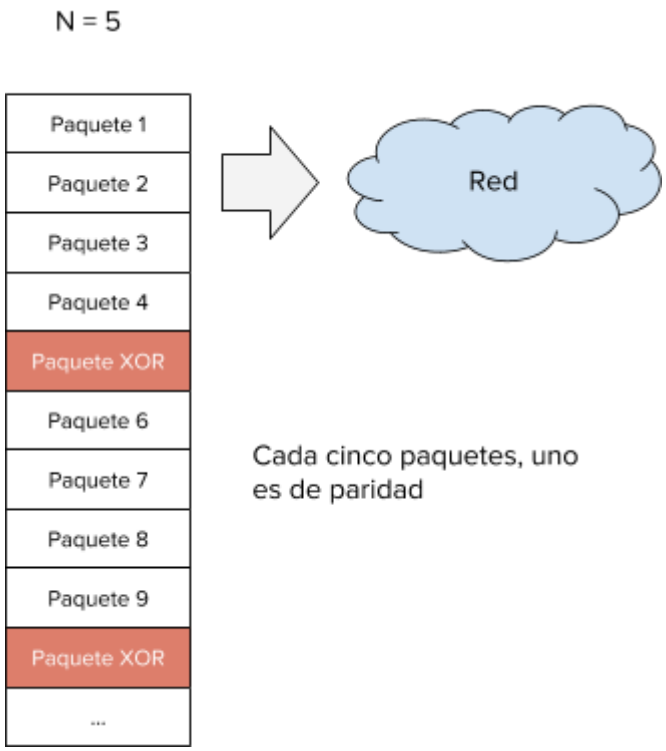


Figura 7.4.1: Generación de paquetes XOR

Cada cuántos paquetes se calcula el paquete de paridad es modificable en las constantes de la aplicación (*Constante.Configuracion.numPaquetesParidad*).

El cálculo que se realiza para obtener el paquete de paridad es muy sencillo y consiste en realizar la **OR exclusiva** de la información de audio contenida en los **n-1** paquetes anteriores, siendo **n** la constante que indica el número de paquetes de los cuales uno debe de ser de paridad.

Por ejemplo, si tenemos la constante *numPaquetesParidad* a 4, se enviará tres paquetes con datos de audio y un cuarto con la paridad resultante de la operación XOR de los paquetes anteriores. Veamos un ejemplo ilustrativo:

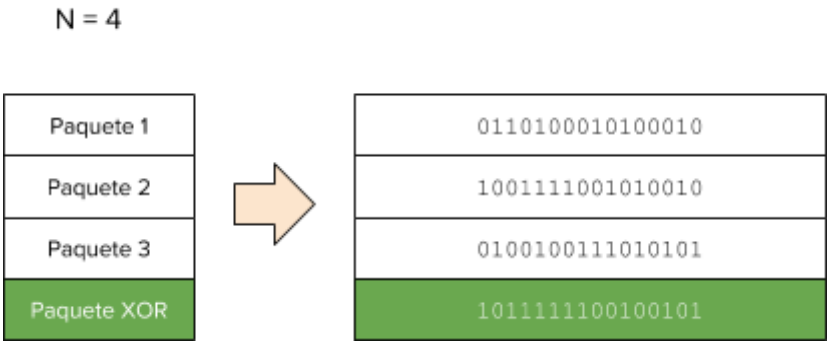


Figura 7.4.2: Generación del paquete XOR cuando n = 4

De esta manera, podremos recuperar uno de los paquetes anteriores al de paridad en caso de ser erróneo. Vamos a demostrarlo con la siguiente ilustración:

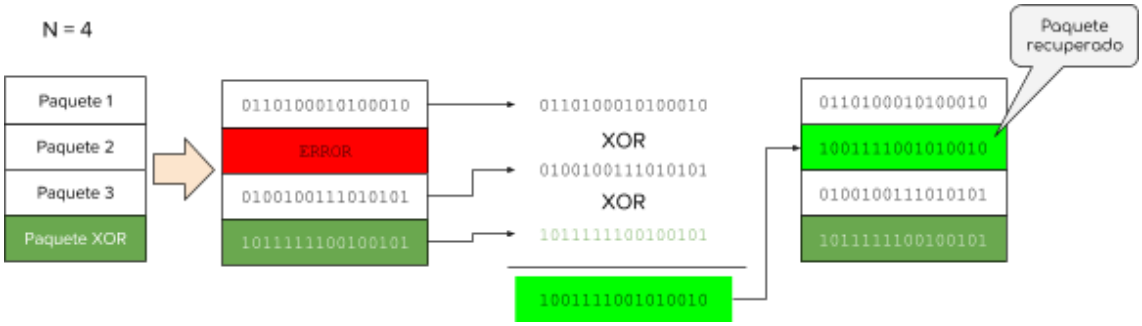


Figura 7.4.3: Generación del paquete XOR cuando n = 4

Como se muestra en la *figura 7.4.3*, vemos que, de una manera muy sencilla, podemos recuperar un paquete y podemos comprobar que la cadena de bits coincide con el original que vemos en la *figura 7.4.2*. En caso de perder dos paquetes del mismo grupo, la recuperación no es posible y se introducirá un silencio.

Es importante destacar que, si el número de paquetes del **buffer de entrada o de contención** es inferior al número de paquetes que deben procesarse para calcular el de paridad, este último no se podrá ser calculado en el destino, pues no dispone de los paquetes necesarios como para poder calcularlo, con lo cual, no podremos recuperar uno de los **n-1** paquetes anteriores en caso de ser erróneo.

En la siguiente figura, podemos observar el caso de tener un buffer de contención inferior a **n**, siendo **n** el número de paquetes que se procesan para calcular el de paridad.

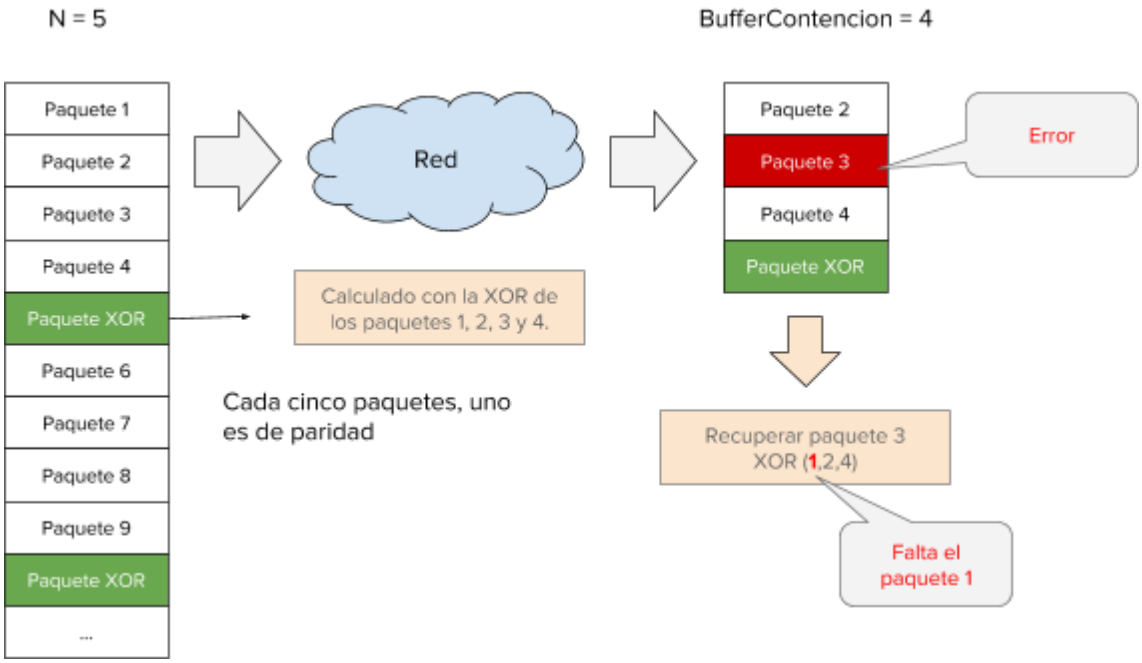


Figura 7.4.4: Buffer inferior al número de paquetes para crear el de paridad (XOR)

Cuando la aplicación detecta que se recibe un paquete de paridad, esta ejecuta los procesos necesarios para verificar si hay un paquete defectuoso. En tal caso, regenerará el paquete con la XOR de los demás del grupo.

Existe una clase tanto en la aplicación de escritorio como en la aplicación móvil, llamada *ControlFlujo*, que contiene los procedimientos de control y regeneración de paquetes.

8. Arquitectura

La arquitectura de la aplicación, es compatible con una estructura *Modelo – Vista – Controlador*. No hay mucho que decir de este patrón tan ampliamente utilizado. Esta arquitectura, nos permite separar los elementos relacionados con la presentación de los datos, los modelos que representan las entidades utilizadas y la lógica de la aplicación que hemos llamado **Servicio**. El servicio es el encargado de implementar la funcionalidad de la aplicación y supone un enlace entre la vista y el modelo.

Además, se ha agregado la capa de acceso a datos con el fin de separar la implementación relativa al acceso a los datos de la lógica de aplicación. De esta manera, además de disponer de una aplicación escalable, si necesitáramos almacenar la información en otro medio o de otra forma distinta, como en una base de datos, únicamente tendríamos que modificar los objetos afectados de esta capa.

Los proyectos se crean con la herramienta de desarrollo **Eclipse**. En el caso de la versión de Android, existe además un módulo o *plugin* de **Eclipse** *Android Development Tools* (ADT) y el *Software Development Kit* (SDK) que incluye un conjunto de herramientas de desarrollo para el sistema Android.

En el caso de la aplicación de escritorio, se ha creado un proyecto *Java* estándar al que hemos añadido una librería externa *org.json.jar* que nos facilita la gestión de datos en formato JSON.

En cuanto a la versión de Android, se ha creado un proyecto *Android Application Project* con las siguientes características:

- **Versión mínima** requerida de la API SDK: API 8 - Android 2.2 (Froyo).
- ***Target SDK*** a utilizar en el desarrollo: API 9 - Android 2.3 (Gingerbread).
- **SDK de compilación**: Vamos a utilizar la versión API 20 - Android 4.4 (KitKat) para compilar el programa.

La versión mínima nos permite especificar cuál es la versión del sistema operativo del dispositivo a partir de la cual la aplicación es compatible. De esta manera, un dispositivo puede detectar si la aplicación funciona con la versión de su sistema o no.

La versión *Target SDK* utilizada en el desarrollo, es especialmente útil para indicar para qué versión está pensada y testeada la aplicación. Esto nos permite crear la aplicación compatible con futuras versiones del sistema. De esta manera Android puede cambiar el comportamiento en cuanto a la ejecución de la aplicación si la versión *Target SDK* es inferior a la del dispositivo donde se ejecuta.

El SDK de compilación normalmente es superior a los parámetros anteriores y no tiene por qué coincidir con los mismos, pues simplemente indica cual es el SDK instalado para la compilación.
Una vez configurados estos parámetros, Eclipse es capaz de detectar si estamos utilizando clases, funciones o métodos incompatibles con la versión mínima del SDK que estamos utilizando.

Dado que vamos a utilizar un código que sea bastante genérico, hemos seleccionado un SDK lo suficientemente antiguo como para garantizar que la aplicación funcione en dispositivos modernos.

8.1. Conceptos generales

La aplicación implementa tanto la parte del ***cliente activo*** como la del ***cliente pasivo***. Es decir los clientes utilizan la misma aplicación tanto para conectar activamente como para escuchar conexiones.

Es necesario que uno de los clientes escuche, para poder establecer la conexión con él. Una vez establecida la conexión, tanto el cliente activo como el pasivo, generan dos hilos cada uno. Un hilo de entrada que recibe el audio del contrario y lo envía al altavoz del dispositivo local y otro de salida que envía el audio del micrófono de la máquina local por la misma conexión al destino.

Como hemos comentado anteriormente, se utilizarán 16 bits de resolución por muestra y 44100 muestras por segundo, utilizando un sólo canal para limitar la cantidad de información que se envía y se recibe.

Utilizamos estos valores por su amplia compatibilidad con los dispositivos móviles. No todos los dispositivos pueden utilizar frecuencias de muestreo inferiores o superiores.

8.2. Consideraciones de implementación

Con el fin de realizar procedimientos no bloqueantes para la aplicación, se ha optado por implementar un sistema de funcionamiento a través de hilos de ejecución. Esto nos permite optimizar los recursos de la aplicación y realizar operaciones paralelas. Sin embargo, la complejidad aumenta considerablemente en el desarrollo de aplicaciones de este tipo, dado que debemos tener en cuenta que pueden producirse accesos concurrentes a recursos. Esto nos condiciona y nos obliga a implementar sistemas de acceso ordenado como son semáforos en la ejecución del programa.

Los semáforos son sistemas para restringir o permitir el acceso a recursos compartidos de la aplicación. En la siguiente figura, podemos visualizar como el buffer de entrada y

salida son espacios críticos que son accedidos concurrentemente por diferentes procesos.

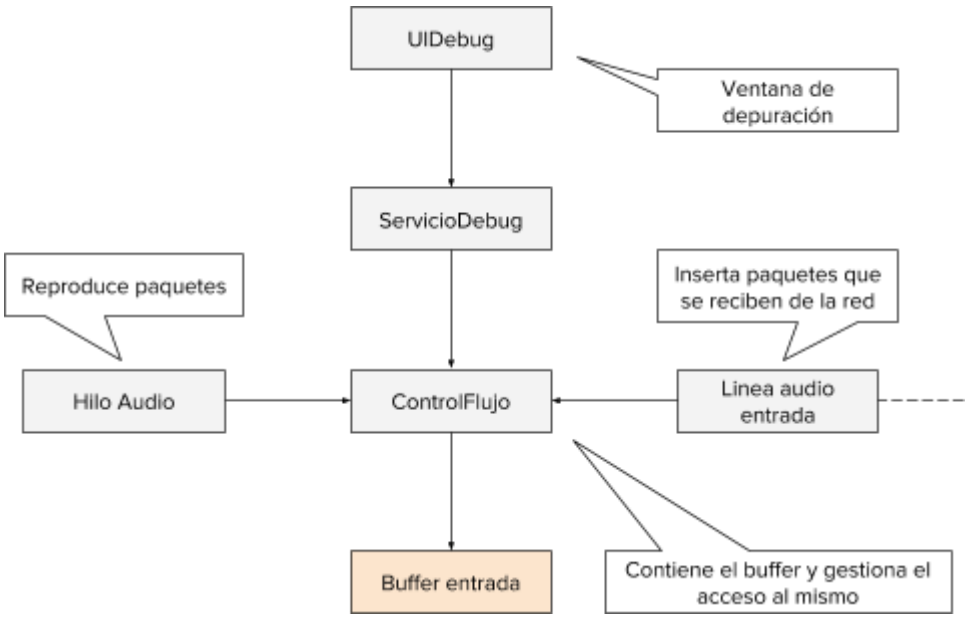


Figura 8.2.1: Acceso concurrente al buffer de entrada

Como observamos en la figura, el acceso al buffer de entrada se realiza a través de la clase *ControlFlujo*. Esta clase es la que contiene el buffer en forma de mapa de objetos y dispone de métodos para verificar paquetes, crear paquetes de audio, insertar paquetes y permite generar los paquetes de paridad (XOR).

Dado que cada uno de los accesos se realiza en hilos diferentes, el acceso al mapa del buffer puede realizarse concurrentemente lo que provoca un error de acceso concurrente en la aplicación.

Con el objeto de poder realizar las operaciones de forma paralela se ha implementado un método bloqueante al control de flujo para que cada hilo pueda obtener un *relevo*, bloqueando el acceso al mismo y disponiendo del control de la gestión de flujo, dejando al resto de hilos suspendidos hasta que lo desbloquee. Cuando el hilo que bloquea el acceso termina su gestión, lo liberará, momento en el que el primero que acceda, dispondrá de ese *relevo*.

Esta funcionalidad se ha realizado a través de un método *setBloqueo()* de tipo *synchronized* que, en Java, permiten que no sea posible la invocación concurrente del mismo [15], de manera que cuando un hilo está ejecutando el método, los otros hilos que lo invoquen permanecerán bloqueados hasta que el primer hilo termine con el objeto.

Pero queda otro asunto que tratar, pues una cosa es el acceso concurrente al método y otra es el acceso al objeto. Nos interesa bloquear el acceso al objeto para que no se realicen operaciones con el mapa de paquetes de entrada. Para ello hemos implementado una estrategia que consiste en provocar que cuando un hilo que ha

ejecutado el método sincronizado detecte que está bloqueado, espere hasta que se desbloquee. Esto es posible llevarlo a cabo mediante la orden *wait()*.

Si el hilo que lo tenía bloqueado vuelve a llamar al método para **liberar** el objeto, se enviará una notificación al resto de hilos para que puedan dejar de estar suspendidos y continuar su ejecución. Esto se realiza mediante la orden *notifyAll()*.

Estas estrategias de bloqueo, se han implementado en la clase *ControlFlujo*, evitando, de esta manera, los errores de concurrencia al buffer.

Veamos el método en cuestión:

```
public synchronized void setBloqueo(Boolean bloquear) throws InterruptedException
{
        //>> Si se quiere bloquear, hay que esperar si está bloqueado
        if (bloquear)
        {
                while (bloqueado)
                {
                        wait(); //>> El hilo deberá esperar
                }
        }
        else
        {
                notifyAll();
        }
        bloqueado = bloquear;
}
```

Figura 8.2.2: Función sincronizada para la implementación de semáforos de un hilo (cerrojos)

Dado que la clase *Conexion* también puede experimentar concurrencia, se han tenido que realizar estrategias similares. Hemos configurado el método *inicializarSocket* para que disponga de bloqueo a concurrencia mediante la orden *synchronized*.

```
public synchronized void inicializarSocket(Boolean activo) throws Exception
{
        try
        {
                if (conectado()) return;
                socket = new DatagramSocket(puertoEscucha);
                socket.setSendBufferSize(Constante.Configuracion.bufferRecepcion);
                socket.setReceiveBufferSize(Constante.Configuracion.bufferRecepcion);

                if (puertoEscucha == Constante.Configuracion.puertoAudio)
                {
                        servicio.insertarLog("Puerto de escucha Audio: " +  puertoEscucha);
                }

                if (puertoEscucha == Constante.Configuracion.puertoAudio)
                {
                        servicio.insertarLog("...");
                }
```

```
        }
        catch (Exception e)
        {
                throw e;
        }
}
```

Figura 8.2.3: Método inicializarSocket con cerrojo

9. Implementación aplicación de escritorio

Vamos a describir la estructura de implementación que se utiliza en la aplicación de **escritorio**. Esta implementación resulta especialmente importante dado que representa la base donde se desarrolla la versión de móvil.

9.1. Estructura del proyecto

Existen, por tanto, cuatro secciones principales que, dado que estamos empleando Java, hemos separado en diferentes paquetes. Dentro de cada uno de estos paquetes residen las clases de la aplicación y otros subpaquetes para organizar el código.

DAO	**JJ.app.glitchy.dao**	
Modelo	**JJ.app.glitchy.modelo**	
	JJ.app.glitchy.modelo.audio	Clases de audio
	JJ.app.glitchy.modelo.contacto	Clases relacionadas con Contactos
	JJ.app.glitchy.modelo.nucleo	Clases que conforman el núcleo como clases de las que heredan o dan soporte a otras.
	JJ.app.glitchy.modelo.debug	Clases de depuración
Controlador	**JJ.app.glitchy.servicio**	
	JJ.app.glitchy.servicio.debug	Herramientas de depuración
	JJ.app.glitchy.servicio.nucleo	Servicios de soporte a otros
Vista	**JJ.app.glitchy.ui**	
	JJ.app.glitchy.ui.cmp	Clases de soporte a componentes de interfaz
	JJ.app.glitchy.ui.debug	Interfaz de depuración

Tabla 9.1.1: Estructura de los paquetes

A continuación vamos a mostrar las clases que conforman la aplicación respetando la estructura del proyecto. Más adelante describiremos algunas de las más importantes.

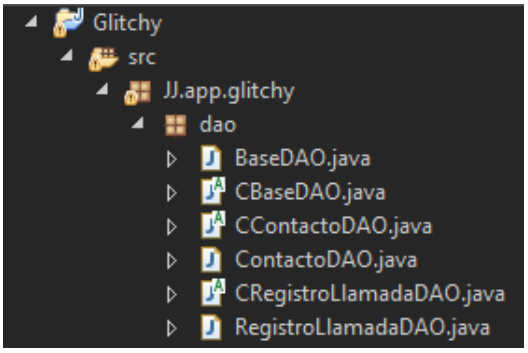


Figura 9.1.1: Clases de la capa DAO

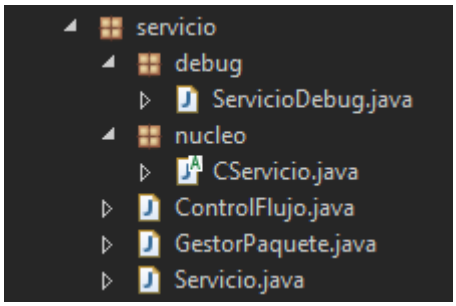


Figura 9.1.2: Clases de la capa servicio

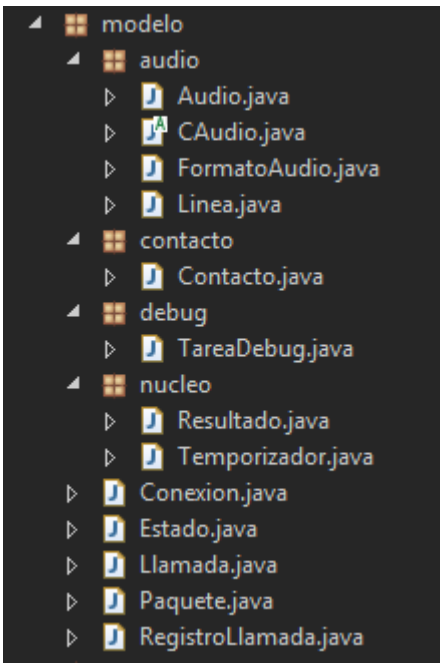


Figura 9.1.3: Clases de la capa modelo

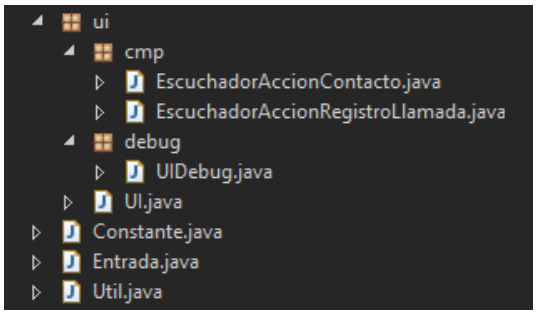


Figura 9.1.4: Clases de la capa vista

Como podemos observar en las figuras, aparecen diferentes clases abstractas. Estas clases nos permiten definir, a su vez, métodos abstractos que implementarán las clases específicas de la plataforma en cuestión y que comentaremos más adelante.

9.1.1. Clases principales del sistema

A continuación comentamos las principales clases que componen la aplicación:

- **Entrada**: Clase principal con el método de entrada *main()*.
- **UI (*User Interface*)**: Esta clase agrupa las características de configuración de la ventana (botones, listados, cuadros de texto, etc.). En esta clase no está la lógica

de la aplicación, tan solo los eventos y la parte de interfaz de usuario. Se encontraría ubicado en la capa de **vista** en un modelo MVC. Se trata básicamente del punto de conexión entre el usuario y la funcionalidad de la aplicación (controlador).

- **Servicio**: Esta clase representa el controlador principal de la aplicación. Correspondería al servicio **controlador** donde reside la lógica de la aplicación. Implementa los métodos que utiliza las clases del resto de la aplicación para llevar a cabo una o varias tareas específicas.

- **Conexión**: Esta clase contiene la configuración de la conexión y permite conectar con el destino. Tiene como atributos parámetros de configuración de la IP, puerto, socket y los códigos de petición de conexión y respuesta para establecer la comunicación que se lleva a cabo entre terminales. Esta clase es genérica y se utiliza en la de móvil mediante una relación de herencia.

- **FormatoAudio**: Nos permite definir de una manera genérica el formato de audio que gestiona la aplicación: contiene atributos como la frecuencia de muestreo o *sample rate*, la resolución de muestra o tamaño de muestra (8, 16 bits), el número de canales, si los valores de muestra son con signo o sin signo, así como el orden en el que se guarda la información (*big-endian*, *little-endian*). Esta clase es genérica y se utiliza igual tanto en la versión de escritorio como en la de móvil.

- **ControlFlujo**: Permite gestionar la información que se guarda en el buffer así como el formato con el que se debe almacenar. También contiene el buffer de paquetes y las estrategias de control de flujo para solucionar problemas de congestión. Se trata de una clase de núcleo y es la misma tanto en la aplicación de escritorio como en la de móvil.

- **Linea**: Esta clase representa el controlador de la salida y recepción de audio que se transmite entre dispositivos. Contiene una referencia a la clase *Conexion* para saber cuál es el destinatario. También dispone de una referencia a la clase *Audio* que se implementa y depende de la plataforma, pues no es lo mismo acceder al micrófono desde un PC que desde un terminal móvil.
  Además, implementa la interfaz *Runnable* para poder crear hilos de ejecución, uno para la entrada de audio y otro para la salida. Esta clase se utiliza en la versión móvil mediante una relación de herencia.

- **Audio**: Esta clase es la encargada de gestionar la lectura y reproducción del audio. Es **dependiente de la plataforma**, es por ello que se ha implementado una clase abstracta *CAudio* de la que se hereda para que otras plataformas puedan implementar su propia clase audio.

- **Util**: Esta clase es transversal a la aplicación y dispone de métodos y herramientas genéricas independientes que da soporte al resto de la aplicación. Contiene métodos estáticos de los que se puede hacer uso sin necesidad de

instanciarla. Esta clase es genérica y es la misma tanto en la versión de escritorio como en la de móvil.

➔ **Constante**: Al igual que *Util* esta clase es transversal a la aplicación y contiene las constantes de la aplicación como el tamaño del buffer de contención, los estados de la aplicación, número máximo de secuencia para la identificación de los paquetes, texto de petición y respuesta de conexión, tiempo de espera en la escucha, etc. Contiene variables estáticas con el fin de poder leerlas desde cualquier parte de la aplicación. Esta clase se utiliza en la aplicación móvil mediante una relación de herencia.

La estructura básica de las clases más destacadas del proyecto y la relación entre las mismas, es la siguiente:

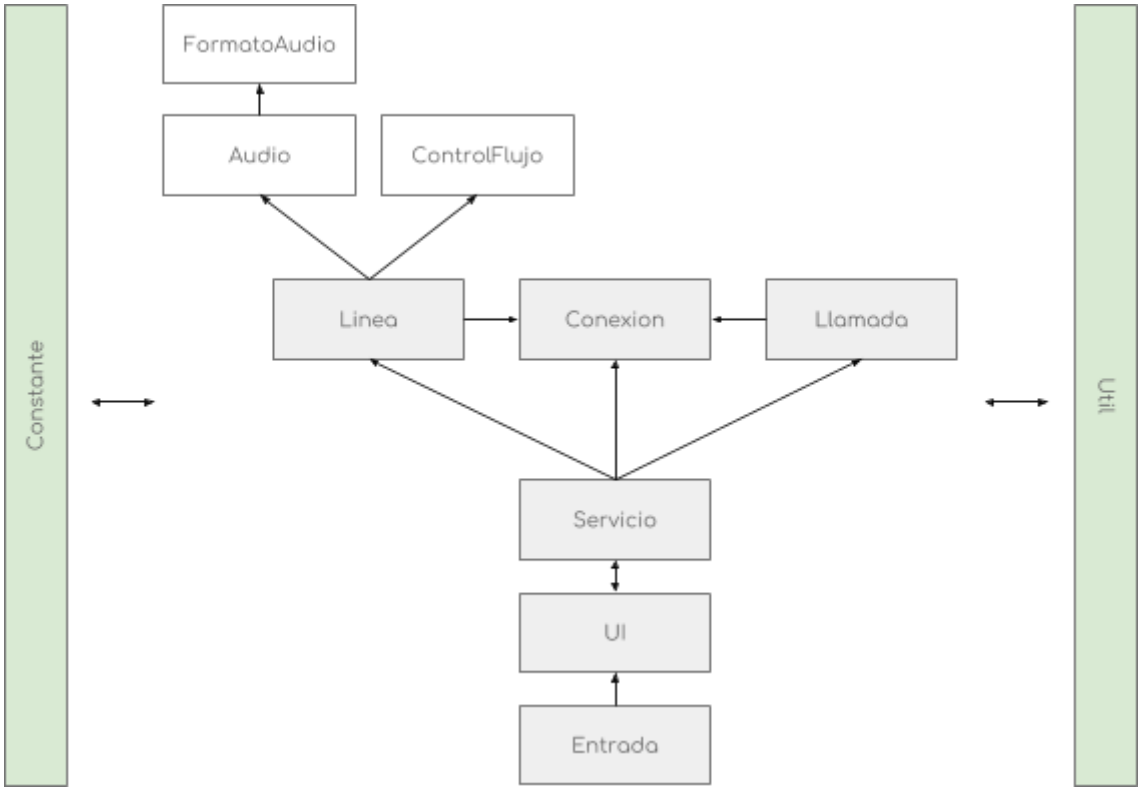


Figura 9.1.1.1: Estructura de clases general de la aplicación

El motivo de instanciar la clase *Conexión* en el servicio es que, además de que el servicio gestiona las conexiones, esta clase es utilizada por *Linea* y *Llamada* cuando se lleva a cabo una llamada.

9.1.2. Capa de acceso a datos

Con el fin de separar la lógica de la aplicación de los métodos relacionados con el acceso a datos, se ha implementado una capa dedicada a tal fin.

El objetivo de esta capa es construir código bien organizado, legible y mantenible, así como reutilizar código con el fin de aumentar la escalabilidad del proyecto.

En esta capa existen algunas clases abstractas con el fin de personalizar la implementación en otras plataformas. Más adelante, detallaremos cada una de ellas.

9.1.3. Estructura y funcionalidad del servicio

El servicio es el encargado de crear el hilo para atender una nueva conexión. Por tanto, dispone de un objeto de tipo *Conexion* que se configura cuando se negocia una nueva conexión realizada por un cliente. Una vez se lleve a cabo este proceso, esta conexión será la que utilizará la clase *Linea* que, a través de la clase *Estado*, permitirá monitorizar si un usuario cuelga la llamada.

Cada flujo se desarrolla en un hilo de ejecución.

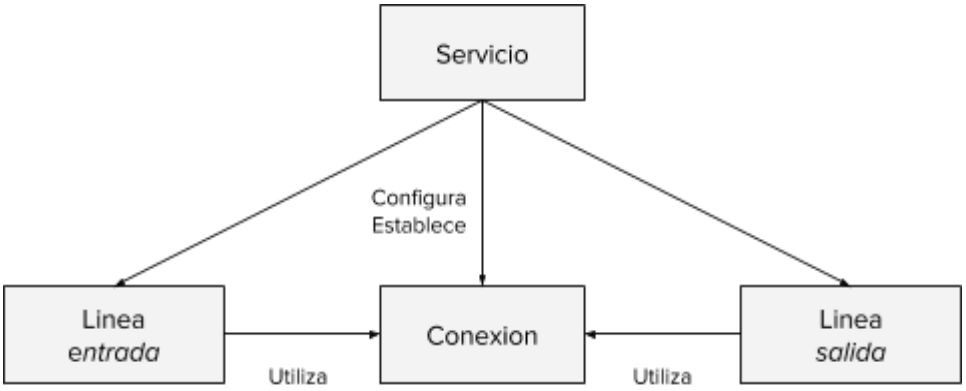


Figura 9.1.3.1: Funcionamiento gestión audio del servicio

Para verlo más claramente, vamos a resumir los pasos que se realizan para realizar la conexión.

1. Al pulsar el botón *escuchar*, el servicio crea un hilo de ejecución para que la aplicación permanezca a la escucha por el puerto especificado.
2. Si alguien conecta, se obtiene su dirección IP y puerto efímero que el cliente ha abierto y se configura el objeto *Conexion*.
3. El servicio crea los hilos de audio de entrada y de salida a partir de la conexión configurada. Es decir, el servicio inicializará los objetos *Linea* pasándole la referencia de la **conexión** que va a utilizar para enviar y recibir la información (en bytes). La conexión representa el enlace entre terminales. El objeto *Conexion* tiene todo lo necesario para poder enviar o recibir información. Dispone de una dirección IP, un puerto y el socket de conexión entre los clientes. Por ese socket se puede enviar y recibir información.

Debemos comentar que, tal y como se ha indicado anteriormente, realmente existen dos instancias de la clase *Linea*, por parte del servicio, una para la entrada de audio y otra para salida del **flujo de datos**.

La conexión se utilizará, inicialmente, para establecer la conexión, la llamada e intercambiar mensajes relacionados con el estado de la llamada. Es decir, mientras se está entablando una comunicación de audio, es posible enviar mensajes de estado entre los cuales estaría el de poder colgar la llamada por cualquiera de las dos partes.

9.2. Consideraciones de implementación

Dado que cada plataforma dispone de su propia implementación, hemos optado por llevar a cabo una estrategia que consiste en desarrollar clases abstractas que nos faciliten la creación de clases específicas que implementen los métodos necesarios para llevar a cabo la solución.

A continuación, vamos a listar las clases que nos servirán de base para la implementación de la versión de Android y nos permitirán desarrollar una implementación dependiente de la plataforma.

Evidentemente, si la plataforma está desarrollada en otro lenguaje de programación, esta estrategia requeriría de herramientas intermediarias que permitan interpretar clases Java para generar código nativo. Como hemos comentado anteriormente, Oracle, actualmente propietaria de Java, dispone de un *framework* Oracle Mobile Application Framework que permite realizar aplicaciones móviles utilizando el lenguaje Java.

CAudio	Clase abstracta que permite gestionar el tratamiento del audio.
CServicio	Clase de la que hereda los servicios de la aplicación con los métodos comunes a todos ellos.
CBaseDAO	Clase de la que heredan los DAO específicos con los métodos comunes a los mismos.
CContactoDAO	Clase de la capa DAO para englobar los métodos de acceso a datos relacionados con los contactos y separarlos de la lógica de la aplicación.
CRegistroLlamadaDAO	Clase de la capa DAO para englobar los métodos de acceso a datos relacionados con el registro de las llamadas.

Tabla 9.2.1: Clases abstractas

La clase **CAudio**:

Si nos fijamos en la *figura 9.1.3*, en el apartado "9.1. Estructura del proyecto", podemos observar que en el paquete *JJ.app.glitchy.modelo.audio* existe la clase *Audio* y *CAudio*. La primera, hereda de *CAudio* e implementa los métodos abstractos de la misma.

CAudio no es una interfaz ya que buscamos una clase que represente al audio del sistema y de la que hereden las implementaciones específicas de la plataforma. De esta manera las nuevas clases ya dispondrán de los atributos necesarios para representar la clase que gestiona el tratamiento de audio. Por tanto, *CAudio* y *Audio* tienen una relación de dependencia bastante clara tal y como veremos más adelante.

```
public abstract class CAudio implements Runnable
{
	protected FormatoAudio formato;
	protected CServicio servicio;
	protected ControlFlujo flujo;
	protected Boolean activado;
	protected Boolean reproducir = false;

	public CAudio(CServicio servicio)
	{
		this.servicio = servicio;
		formato = getFormatoAudio();
	}

	public abstract void inicializar();
	public abstract void iniciarMicrofono();
	public abstract void pararMicrofono();
	public abstract ByteBuffer leerMicrofono(byte[] bloque);

	public void activar(Boolean b)
	{
		activado = b;
	}

	@Override
	public void run()
	{
	}

	//>> Operaciones de muestreo e interpolación
	...
	private FormatoAudio getFormatoAudio()
	{
		float sampleRate = 44100f;
		int sampleSizeBits = 16;
		int channels = 1;
		boolean signed = true;
		boolean bigEndian = false;
		FormatoAudio f = new FormatoAudio((int)sampleRate, sampleSizeBits,
channels);
		f.setSigned(signed);
		f.setBigEndian(bigEndian);
		return f;
	}
	//>> Métodos GET / SET
	...
}
```

Figura 9.2.1: Clase abstracta CAudio

La clase **CServicio**:

Esta clase abstracta, nos permite crear nuevos servicios que, además de contener los atributos comunes de un servicio, nos permite hacer uso de la funcionalidad del núcleo de la aplicación. Es decir, dado que la aplicación se ha diseñado para servir como librería tanto de clases como de funcionalidad, simplemente heredando de *CServicio* ya disponemos de la funcionalidad básica utilizada en la aplicación de escritorio.

La clase *CServicio* dispone de métodos abstractos que la aplicación móvil deberá implementar dado que son específicos de la plataforma.

```
public abstract class CServicio
{
        public String nombre = Constante.nombre;        //>> Nombre de la aplicación
        public String version = Constante.version;      //>> Versión de la aplicación
        protected String log = "";                      //>> Log interno de depuración
        CBaseDAO dao;
        CContactoDAO contactoDAO;
        CRegistroLlamadaDAO registroLlamadaDAO;

        //>> Métodos de asignación de DAO
        ...

        public abstract void insertarLog(String txt);
        public abstract void setTxtEstado(String txt);
        public abstract void setEstadoBuffer(Integer b);
        public abstract void setEstadoMic(Integer b);
        public abstract void setEstado(Integer v);
        public abstract void setTxtDebug(String txt);

        //>> Métodos CRUD de gestión de contactos
        ...
        //>> Métodos de gestión de registros llamadas
        ...
}
```

Figura 9.2.2: Clase abstracta CServicio

La clase **CBaseDAO**:

Esta clase contiene los atributos básicos que todo DAO debería de tener. En nuestro caso, dispone de los métodos de acceso al medio que deberán sobrescribir las clases de la plataforma, que hereden de la misma. Entre sus métodos nos encontramos con *leerDatos* y *guardarDatos*. Se trata de una clase base de la que heredan los DAO.

```
public abstract class CBaseDAO
```


```
{
	protected JSONObject datos;
	protected String seccion = "datos";

	protected abstract JSONObject leerDatos();
	public abstract Boolean guardarDatos(JSONObject datos);
}
```

Figura 9.2.3: Clase abstracta CBaseDAO

La clase **CContactoDAO**:

Como su propio nombre indica, esta clase permite realizar las operaciones de acceso a datos relacionadas con contactos. Hereda de *CBaseDAO* y entre sus métodos, nos encontramos con *getContactos*, *crearContacto* y *eliminarContacto*, operaciones típicas que se implementan en este tipo de clases.

La clase **CRegistroLlamadaDAO**:

Al igual que ocurre con *CContactoDAO* esta clase será la encargada de realizar las operaciones CRUD en cuanto a registros de llamada se refiere.

9.3. Datos de la aplicación

La información relacionada con los contactos se almacena en formato **JSON** que nos permite definir estructuras complejas y nos otorga una gran flexibilidad. El fichero utilizado es *datos.json* y tiene una estructura como la siguiente:

```
{
	"datos":
	{
	"contactos": [
		{
			"fecha": "24/08/2018 18:30:05",
			"nombre": "Android",
			"rutaImg": "",
			"uri": "192.168.1.50:20001"
		},
		{
			"fecha": "24/08/2018 05:12:46",
			"nombre": "Tablet",
			"rutaImg": "C:\\Users\\JJ\\Desktop\\Sin título.png",
			"uri": "192.168.1.55:20001"
		}

	],
	"historial": [
		{
			"codigo": "22/08/2018 18:48:30192.168.1.50:54918",
			"fecha": "22/08/2018 18:48:30",
			"nombre": "sabela-pc",
```

```
                        "tipo": "aceptada",
                        "uri": "192.168.1.50:54918"
                },
                {
                        "codigo": "22/08/2018 18:53:26192.168.1.50:20001",
                        "fecha": "22/08/2018 18:53:26",
                        "nombre": "sabela-pc",
                        "tipo": "saliente",
                        "uri": "192.168.1.50:20001"
                }
        ]
}
```

Figura 9.3.1: Ejemplo estructura fichero de datos

9.3.1. Contacto

Hemos implementado una entidad que nos permite almacenar la información relacionada con un destinatario. Se trata de una estructura con los siguientes atributos:

- **Fecha**: Es el día y la hora de la última vez que se actualizó el contacto. Si se crea un contacto nuevo, significará la fecha de creación. Se representa en formato *dd/MM/yyyy HH:mm:ss*.
- **Nombre**: Es un nombre o alias que nos permite identificar el contacto.
- **RutaImg**: Este atributo almacena la ruta de la imagen a mostrar en el contacto.
- **URI**: Se trata de una cadena que identifica de forma única un recurso que en nuestro caso es sencillamente de la dirección IP y el puerto del destinatario. No es necesario indicar el protocolo.

A la hora de insertar un contacto, controlamos si el nombre del mismo ya existe para no añadirlo varias veces. Sin embargo, el sistema nos permitirá agregar un contacto independientemente exista la URI del mismo, en el listado de contactos.

A continuación, mostramos la estructura de la clase Java:

```
public class Contacto
{
        String nombre;
        String uri;
        String rutaImg;
        private Date fecha;

        //>> Métodos GET / SET
        ...
}
```

Figura 9.3.2.1: Clase Contacto

9.3.2. Registro de llamada

Para representar un registro de llamada, hemos creado una estructura con una serie de atributos que la identifican:

- **Codigo**: El código nos permite identificar de forma única un registro de llamada. Es necesario indicar este código para poder hacer referencia a uno en concreto, pues pueden existir varias llamadas del mismo tipo y remitente.
- **Fecha**: Es el día y la hora en la que se realizó la llamada en formato *dd/MM/yyyy HH:mm:ss*.
- **Nombre**: Nos permite identificar el remitente de la llamada. De forma predeterminada este campo contiene la URI de la máquina desde donde se hace la llamada. Sin embargo, si ésta URI existe en la lista de contactos, aparecerá el nombre del contacto.
- **Tipo**: Este atributo almacena el tipo de llamada que puede ser *entrante*, *saliente* o *aceptada*.
- **URI**: Se trata de la dirección IP y el puerto del remitente.

Seguidamente, mostramos la clase Java que identifica un registro de llamada:

```
public class RegistroLlamada
{
        String codigo;
        String nombre;
        String uri;
        String tipo;
        Date fecha;

        //>> Métodos GET / SET
        ...
}
```

Figura 9.3.3.1: Clase RegistroLlamada

9.4. Gestión de audio

Anteriormente, hemos visto cómo se lee, se reproduce y se gestiona a nivel general la información de audio digital para enviarlos a la red. A continuación, vamos a ver qué recursos o clases hemos utilizados para llevarlo a cabo desde el punto de vista de una computadora y un dispositivo móvil de Android.

9.4.1. Gestión de audio en una computadora

En Java existe una clase llamada *AudioFormat* del paquete *javax.sound.sampled* y que utilizamos para asignar este formato de audio y poder trabajar con el mismo.

El acceso al micrófono se lleva a cabo mediante la clase *AudioSystem* que nos da acceso al mezclador o *mixer* instalado en el sistema y contiene una serie de métodos para convertir los datos de audio en diferentes formatos.

También hemos utilizado la clase *TargetDataLine* del paquete *javax.sound.sampled* que nos permite leer la información que se vuelca en la línea del micrófono. Con esta información accedemos al *mixer* para obtener la línea correspondiente. La lectura de los datos a través de *TargetDataLine* consiste en una cadena de bytes que después utilizamos para enviar por el socket.

Esta lógica estará implementada en la clase *Audio*, que hereda de *CAudio* del núcleo de *Glitchy*.

9.4.2. Gestión de audio en Android

En Android el funcionamiento, en cuanto a la gestión del de audio, difiere a la de la aplicación de escritorio. En este caso, se utiliza la clase *AudioTrack* del paquete *android.media.AudioTrack* para enviar el flujo de audio al altavoz del auricular de llamada de voz. Este auricular lo identificamos indicando al constructor de la clase *AudioTrack* la constante *android.media.AudioManager.STREAM_VOICE_CALL* como primer parámetro.

En cuanto a la lectura del audio del micrófono, hacemos uso de *AudioRecord* del paquete *android.media.AudioRecord*. Llama la atención que en Android, para poder leer información del micrófono del dispositivo es necesario realizar una “grabación”.

Al igual que en el caso de la versión de escritorio, esta lógica estará implementada en la clase *Audio* del proyecto que hereda de *CAudio* del núcleo de *Glitchy*.

9.5. Pantalla principal

La pantalla principal consiste en una pequeña ventana con algunos controles para poder gestionar la comunicación, visualizar contactos y acceder al registro de llamadas.

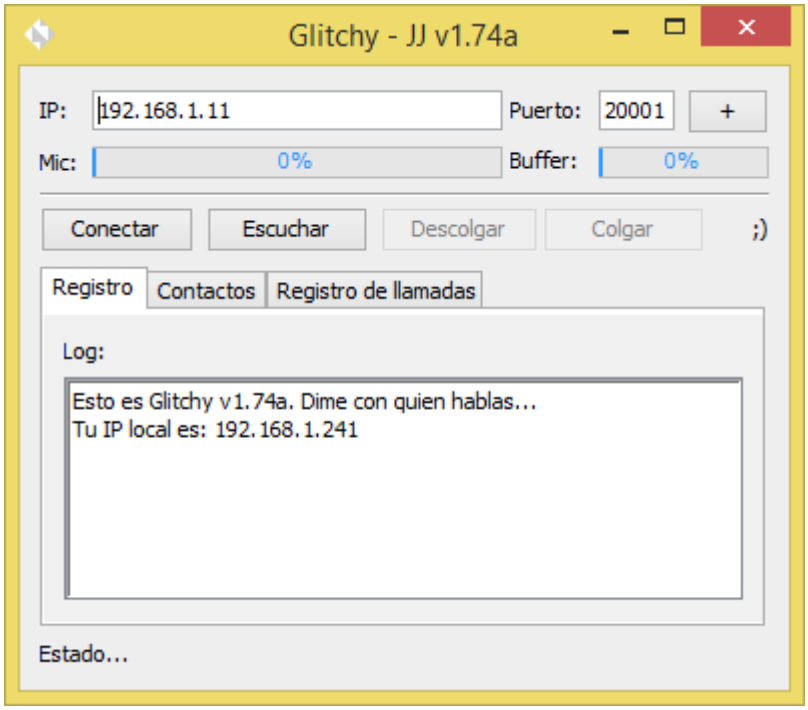


Figura 9.5.1: Pantalla principal aplicación escritorio

Como observamos en la figura anterior, disponemos de controles para poder realizar una llamada a una dirección IP y un puerto entre muchas otras opciones.

En la parte superior de la pantalla están los controles con los que podemos realizar llamadas (botón Conectar), poner el sistema en modo de escucha de llamadas (botón Escuchar), descolgar una llamada entrante (botón Descolgar) y colgar una llamada (botón Colgar). Esta última opción de colgar está disponible siempre que iniciemos una llamada, recibamos una petición de llamada o estemos en una llamada en curso.

Seguidamente, mostramos una imagen con las diferentes opciones de la parte superior de la pantalla. Marcamos en rojo los botones que nos permiten realizar la gestión de llamadas indicados anteriormente.

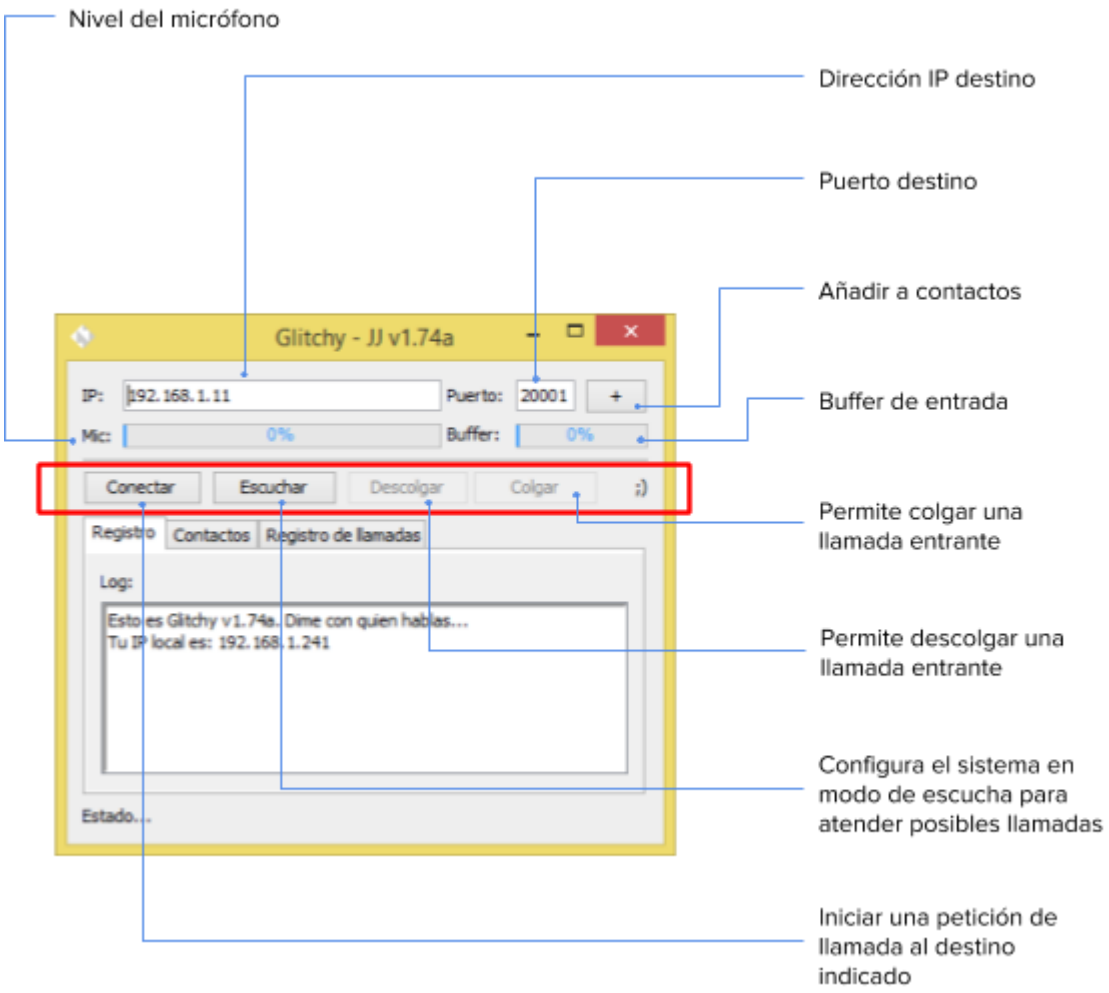


Figura 9.5.2: Controles superiores de la aplicación de escritorio

Como vemos, además de poder atender o realizar una llamada, podemos visualizar el nivel del micrófono y el nivel de buffer de entrada cuando estamos en una llamada activa. Además, también podemos añadir a contactos la dirección IP y puerto que aparecen reflejados en los campos de texto.

Las opciones de la parte inferior están más enfocados a la gestión de los datos de aplicación como son los contactos y el registro de llamadas, así como un log con información de depuración.

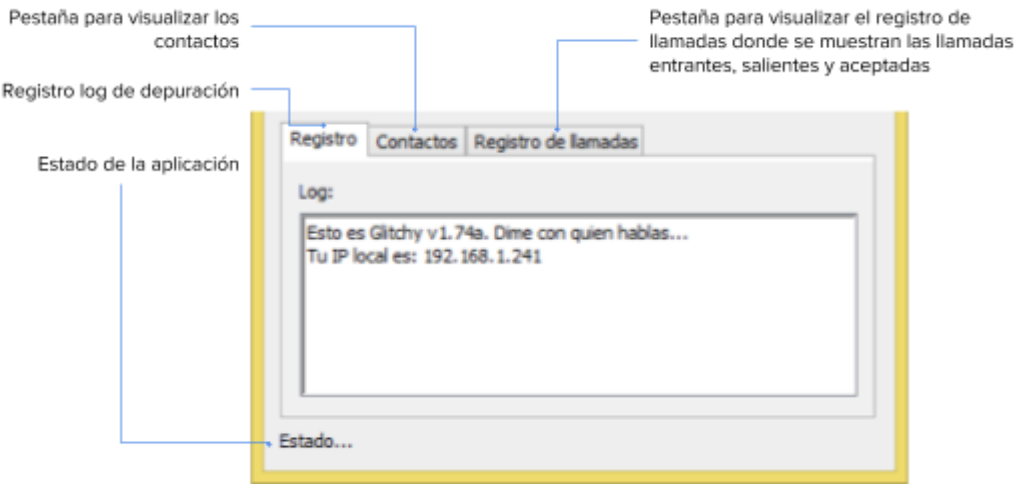


Figura 9.5.3: Controles inferiores de la aplicación de escritorio

Mediante la pantalla de log podemos visualizar cada paso que realiza la aplicación y verificar si el funcionamiento es el esperado.

9.6. Gestión de contactos

Es posible modificar la lista de contactos, agregando o eliminando elementos.

La lista de contactos se encuentra en una pestaña situada al lado derecho de la del registro de log. El contenido de esta pestaña es interactivo, de modo que podemos gestionar los contactos desde el listado que aparece.

En la figura siguiente se puede visualizar el listado de contactos que se muestra al pulsar en la pestaña *Contactos*.

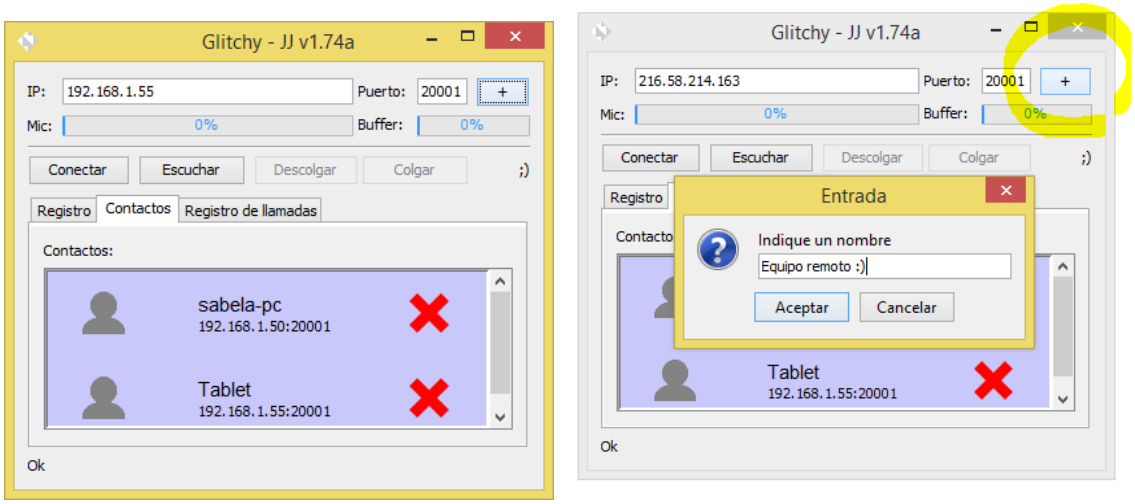


Figura 9.6.1: Pestaña de contactos y botón de agregar contacto

Para agregar un contacto, utilizaremos el botón que aparece en la parte superior con el símbolo más. Al hacerlo, la aplicación nos preguntará el nombre que desea que aparezca en el contacto, como vemos en la figura anterior.

Para eliminar un contacto, basta con pulsar el aspa roja del registro del contacto. La aplicación solicitará la confirmación de eliminación. Es importante destacar que la eliminación de un contacto es permanente, es decir, una vez se elimina el contacto se actualiza la información del medio de almacenamiento.

También es posible modificar la imagen de un contacto pulsando sobre la imagen del mismo. La aplicación nos solicitará la nueva imagen que se desea asignar para posteriormente actualizar la ruta de la misma en el contacto.

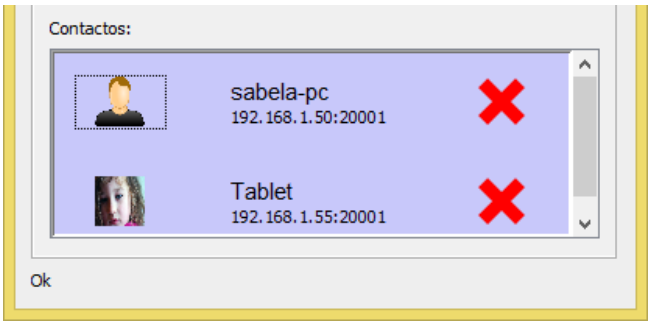


Figura 9.6.2: Modificación imagen de un contacto

9.7. El registro de llamadas

Cada vez que se realiza o se recibe una llamada se registra una entrada en el registro de llamadas.

Podemos consultar el registro de llamadas pulsando sobre la tercera pestaña llamada *Registro de llamadas* de la sección de pestañas de la pantalla principal.

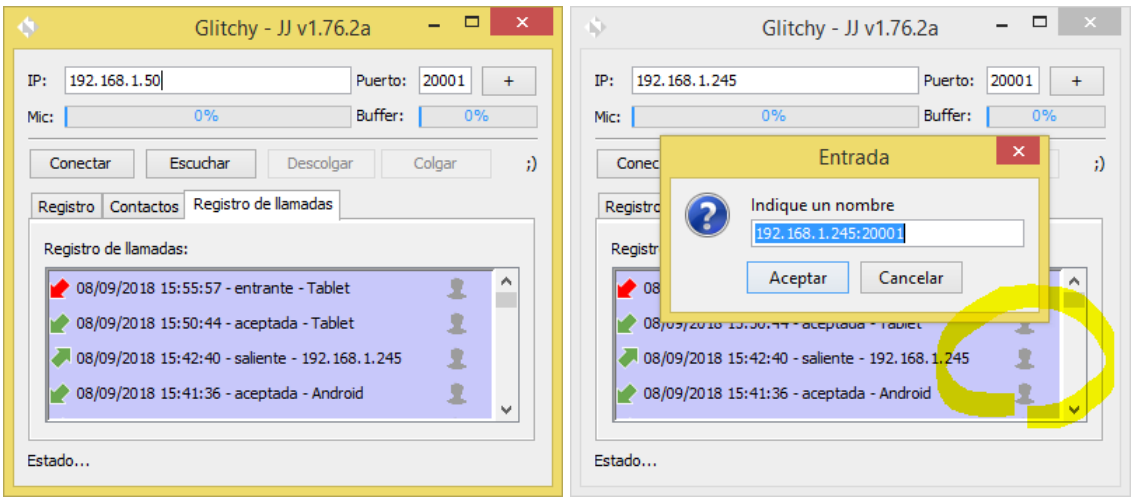


Figura 9.7.1: Pestaña de registro de llamadas y botones de añadir a contactos en el listado

En la figura anterior, podemos visualizar el registro de llamadas desde donde podemos diferenciar distintos tipos de llamada que describimos a continuación:

1. **Entrante o perdida**: Una llamada entrante, es una llamada que se realiza desde un terminal remoto hacia la interfaz de la máquina local. De forma predeterminada, una llamada entrante es una perdida hasta que la misma sea aceptada por el destinatario. Se puede diferenciar este tipo de llamada, mediante un icono en forma de flecha roja que apunta hacia abajo e izquierda.

2. **Saliente**: Una llamada saliente es aquella que se realiza desde la interfaz de red de la máquina local hacia un terminal remoto. Las llamadas salientes se pueden identificar mediante un icono de una flecha verde que apunta hacia arriba y derecha.

3. **Aceptada**: Una llamada aceptada es una llamada entrante que ha sido descolgada por la aplicación local. Este tipo de llamada se representa con una flecha igual que la de la llamada entrante pero de color verde.

En la siguiente figura, podemos visualizar las diferentes llamadas que se han realizado a o desde la aplicación. Podemos observar además que aparece un texto indicando el tipo.

Únicamente se insertan elementos en el registro de llamadas cuando se recibe o se envía una petición de llamada. Por tanto, las llamadas aceptadas son registros de llamada entrante actualizados para indicar que se ha aceptado. Es al descolgar una llamada, cuando se actualiza esa llamada entrante.

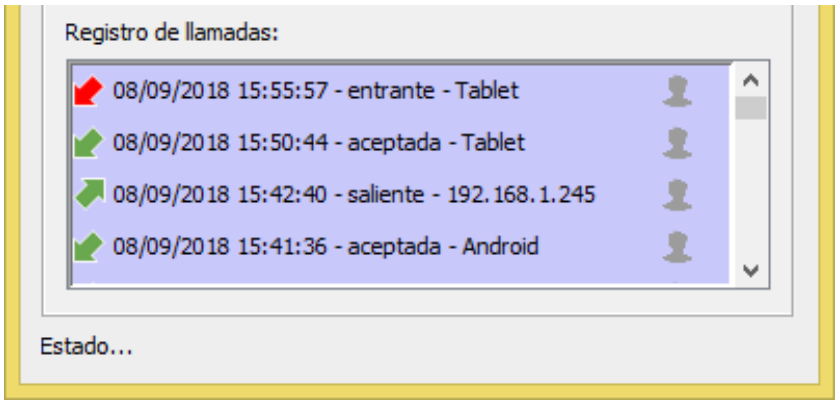


Figura 9.7.2: Tipos de registro de llamada

9.8. Gestión de llamadas

9.8.1. Llamada saliente

Si el usuario desea realizar una llamada deberá pulsar sobre el botón *Conectar* lo que provocará que se lleve a cabo el protocolo de llamada comentado en el punto "6.4 Protocolo de comunicación para realizar una llamada" de esta memoria.

El primer paso que se realizará es intentar conectar con el destino, hecho que veremos reflejado en la pantalla de log con un mensaje indicando que se está intentando establecer conexión con el destino indicado. Si el destino no es alcanzable o la dirección es equivocada, saltará el tiempo de espera que determina si el extremo contesta o no.

Esta primera conexión se realiza a través de la clase *Conexion*, cuya descripción hemos detallado anteriormente y que envía el mensaje "conectar" al destino y quedándose a la espera de respuesta. Para ilustrar esta situación podemos fijarnos en la *figura 6.4.1*.

Si el destino no contesta durante un tiempo determinado, se reiniciará el estado de la aplicación. Este tiempo es modificable a través de las constantes de la aplicación (*Constante.Configuracion.tiempoEspera*).

Si la conexión ha tenido éxito, se procederá con la llamada. Es decir, la aplicación lanzará el mensaje de llamada saliente tal y como se especifica en el protocolo. En esta situación, tanto en el cliente como en el destino se deshabilitan los botones *Conectar* y *Escuchar* quedando disponible el de *Colgar*. Dado que la llamada es saliente desde el punto de vista del usuario que realiza la llamada (activo), únicamente es en la interfaz del destino donde se habilita el botón *Descolgar*.

El usuario activo tiene el botón de colgar habilitado dado que, de esta manera, tendrá la opción de finalizar la llamada sin que necesariamente el otro extremo haya descolgado la llamada.

Esta llamada se realiza a través de un hilo lanzado de la clase *Llamada*. Esta clase es la encargada de enviar el mensaje "llamadaSaliente" al destino y esperar la respuesta.

Al igual que en el caso anterior, si no se recibe respuesta por parte del destino en un tiempo determinado se reinicia el estado de la aplicación. Por defecto, este valor es de 20 segundos aunque es modificable desde las constantes de la aplicación (*Constante.Configuracion.tiempoEsperaLlamada*).

El hecho de colgar la llamada provoca una situación similar a cuando expira el tiempo de espera, se reinicia el estado de la aplicación.

A continuación, mostramos la ventana de la aplicación cuando se realiza una llamada a un terminal.

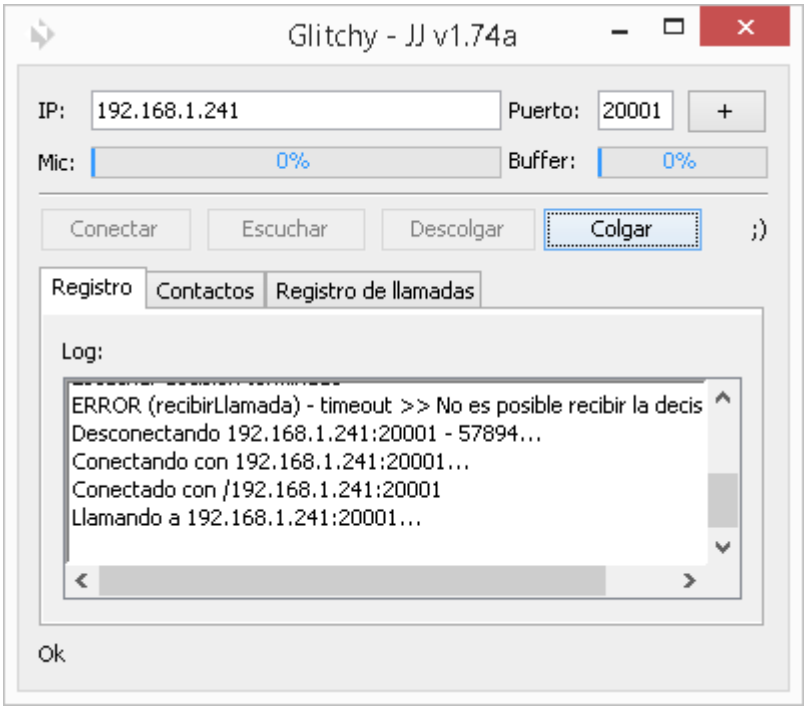


Figura 9.8.1.1: Imagen aplicación de escritorio en estado de llamada saliente

9.8.2. Llamada entrante

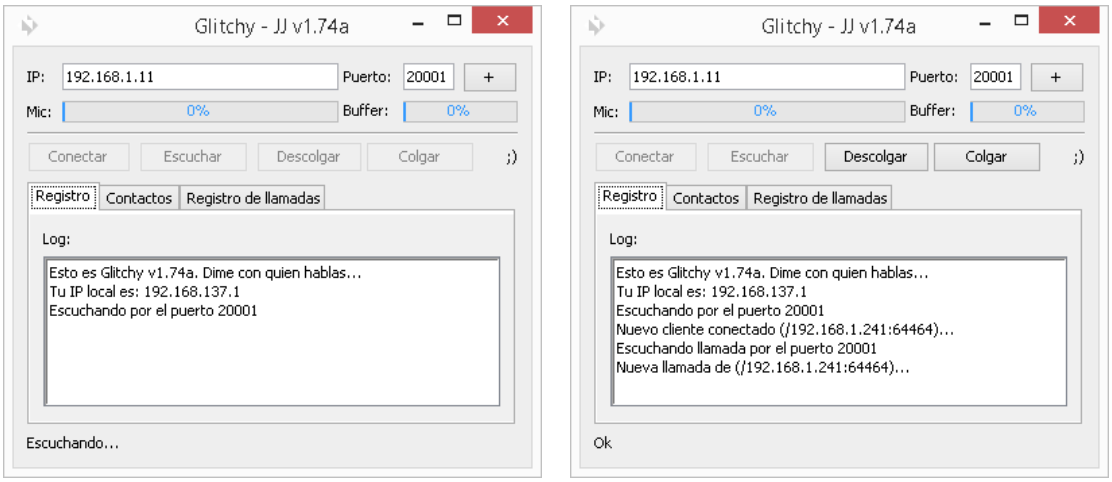
Para poder recibir llamadas es necesario mantenerse a la escucha pulsando el botón *Escuchar*. Cuando un usuario se pone en modo escucha, se deshabilitan todos los botones de gestión de llamadas que son *Conectar*, *Escuchar*, *Descolgar* y *Colgar*.

Una vez uno de los extremos escuche la llamada, cualquier terminal podría iniciar una llamada contra él. Este paso es importante en la aplicación de escritorio dado que los dispositivos no podrán comunicarse si no hay ninguno a la escucha de llamadas.

Si se produce una llamada entrante, aparecerá en el registro log un mensaje indicando esa situación y se habilitarán los botones *Descolgar* y *Colgar* como hemos comentado en el punto anterior teniendo la opción de colgar o descolgar la llamada.

Si se cuelga, se reiniciarán los estados de ambas aplicaciones, pues en el destino se pulsa el botón de colgar que provoca el cambio de estado de la aplicación y además envía el mensaje de control “colgar” al terminal que llama para que reinicie su estado.

Veamos a continuación los diferentes estados de la aplicación de escritorio cuando se escucha y se recibe una llamada:



Escuchando *Recibiendo llamada*

Figura 9.8.2.1: Estado escuchando y recibiendo llamada en la aplicación de escritorio

9.8.3. Llamada en curso

Cuando se lleva a cabo el protocolo de llamada y uno de los terminales descuelga la aplicación pasa al estado de llamada en curso. En este caso, cualquiera de las dos partes tendrá la opción de colgar la llamada.

En esta situación, los hilos de audio están enviando los paquetes de datos de aplicación por la conexión dedicada al audio lo que implica que puedan compartir información a través de señales sonoras. No obstante, a su vez, existe una conexión de estado que las dos partes están atendiendo con el fin de estar pendientes de saber si el otro extremo cuelga o finaliza la llamada. En esta conexión de estado se lanza en un hilo utilizando la clase *Estado* que es la encargada de interpretar este tipo de mensajes.

Por tanto si algún usuario cuelga se enviará el mensaje “colgar” al otro extremo, que recibirá el mensaje y reiniciará su estado.

En la siguiente figura vemos la ventana de la aplicación cuando se está en conversación:

Figura 9.8.3.1: Imagen de la aplicación de escritorio en una llamada en curso

9.9. Ventana de depuración

A pesar de utilizar un entorno de desarrollo tan potente como es Eclipse, cuando se desarrolla una aplicación que trabaja en red y con hilos de ejecución, puede resultar especialmente complicada la depuración y análisis de la información con la que se está trabajando.

Por una parte, la gestión de los paquetes que se envían y se reciben requiere que dispongamos de varias máquinas o varias instancias de la aplicación, una para depurar el cliente activo y otra para depurar el cliente pasivo. Por otra parte, trabajar con varios hilos de ejecución que funcionan de forma paralela, es, como mínimo, algo tedioso de depurar comparándolo con una aplicación con un solo hilo de ejecución.

Debido a lo anterior, surgió la necesidad de implementar la capacidad de visualizar la información de los paquetes que se iban procesando en tiempo real, asunto que se echaba en falta a la hora de verificar el correcto funcionamiento del protocolo y de la gestión de los buffer como el control de flujo y gestión de errores. Para ello, se desarrolló una ventana de depuración orientada a solventar esa falta y, resulta especialmente útil a pesar de no tener un enfoque útil a nivel de usuario que usa a aplicación para realizar llamadas, pues está pensada para fines de desarrollo.

En cuanto a la implementación, las clases que dan soporte a esta ventana están en el paquete *debug* de cada una de las capas de la aplicación, tal y como hemos podido observar en el apartado “Estructura del proyecto”.

Para acceder a esta ventana, pulsaremos sobre el control que muestra el valor del nivel del buffer de entrada.

A continuación, mostramos una imagen de la ventana de depuración y sus controles.

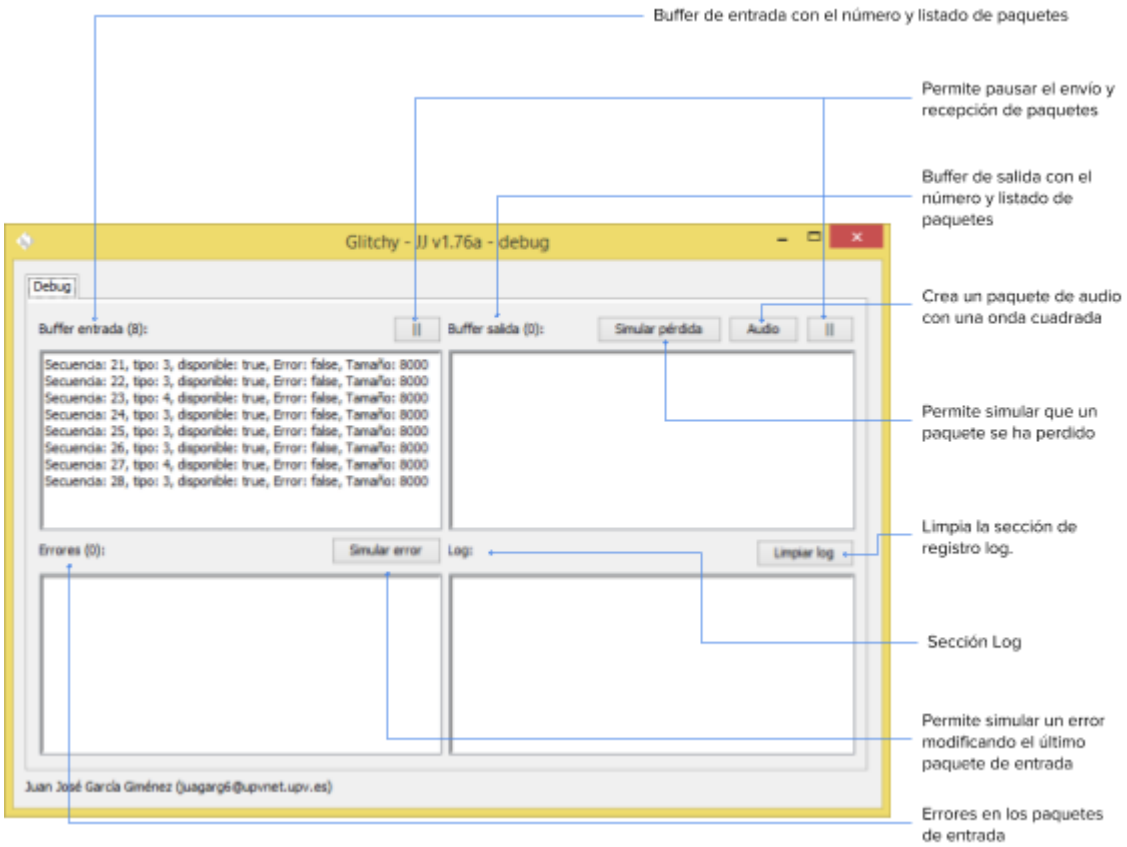


Figura 9.9.1: Ventana de depuración

En esta ventana distinguimos cuatro secciones claramente diferenciadas:

Buffer de entrada: En este apartado se muestra el listado y el número actual de paquetes que lo conforman. Como vemos en la figura también podemos pausar la recepción de paquetes de entrada, de manera que la aplicación, permanecerá si consultar los paquetes entrantes hasta que volvamos a pulsar de nuevo para desactivar la pausa.

Buffer de salida: Esta sección muestra el número actual y el listado de paquetes de salida. Al igual que en la sección anterior, podemos pausar el envío de paquetes de forma que no se enviará nada hasta que volvamos a pulsar el botón.

Esto puede provocar que el destino interprete que la conexión ya no está disponible y que salte el tiempo de espera de recepción de paquetes. Dependerá del tiempo de pausa en el envío de paquetes salientes que salte ese *timeout* establecido en el socket del destinatario.

Errores: Los paquetes con error aparecerán en este listado. Además, dado que estamos en la ventana de depuración, tenemos la opción de simular un error en un paquete mediante el botón *Simular error*. Esto nos permite comprobar si el algoritmo de recuperación de paquetes XOR funciona correctamente.

Log: La sección *Log* nos permite visualizar los registros de estado o errores que se produzcan durante la ejecución de los servicios de depuración.

9.9.1. Simulación de paquete perdido

El funcionamiento de la simulación de pérdida de paquete de la sección de *Buffer salida* consiste en realizar un salto en la secuencia de envío al pulsar el botón *Simular pérdida* con el fin de que el receptor interprete que un paquete no ha llegado e intente recuperarlo. Se trata de una herramienta para testear el funcionamiento del algoritmo de recuperación de errores.

9.9.2. Simulación de errores

El funcionamiento del simulador de error de la sección de *Errores* consiste en que al pulsar el botón *Simular error* se marque como erróneo el último paquete del buffer de entrada para que cuando se lleve a cabo la comprobación de errores, se realicen las acciones oportunas para recuperarlo. De esta manera, cuando el paquete llegue al reproductor, debería haberse recuperado.

Existe la posibilidad de que al pulsar el botón, se intente marcar como erróneo un paquete de tipo paridad, lo que no nos aportaría nada relevante y por tanto, no lo permitimos. Si ocurre este caso, se mostrará un registro de log mostrando el mensaje con la incidencia.

9.9.3. Generación de ondas de prueba

La sección de buffer de salida dispone de un botón que permite generar un paquete de salida con información de audio. Este audio consiste en una onda cuadrada generada con un método que hemos implementado. El motivo de utilizar este control reside en poder testear la aplicación enviando paquetes de audio desde una máquina remota a la que nos conectamos sin necesidad de estar físicamente allí. Además, tampoco es necesarios que el dispositivo remoto disponga de micrófono.

En la siguiente figura vemos el algoritmo que permite realizar la onda cuadrada:

```
public byte[] generar(Integer n, Integer amp)
{
        Integer longitud = 100;
```

```
                int s = 1;
                ByteBuffer bb = ByteBuffer.allocate(n);
                bb.order(ByteOrder.LITTLE_ENDIAN);
                int i = 0;
                while(i < n)
                {
                        if (i % longitud == 0)
                                s = -1 * s;
                        bb.putShort((short)(s * amp));
                        i+=2;
                }
                return bb.array();
        }
```

Figura 9.9.3.1: Método generación onda cuadrada

Como podemos apreciar, es muy sencillo, permite indicar la longitud del *array* resultante y la amplitud. Existe una longitud establecida que indica que cada 100 bytes invierte la onda o, dicho de otra manera, cada 50 muestras se invierte la onda, lo que nos genera una onda cuadrada con un periodo de 2,27 ms y, por tanto, una frecuencia de 441 Hz perfectamente audible.

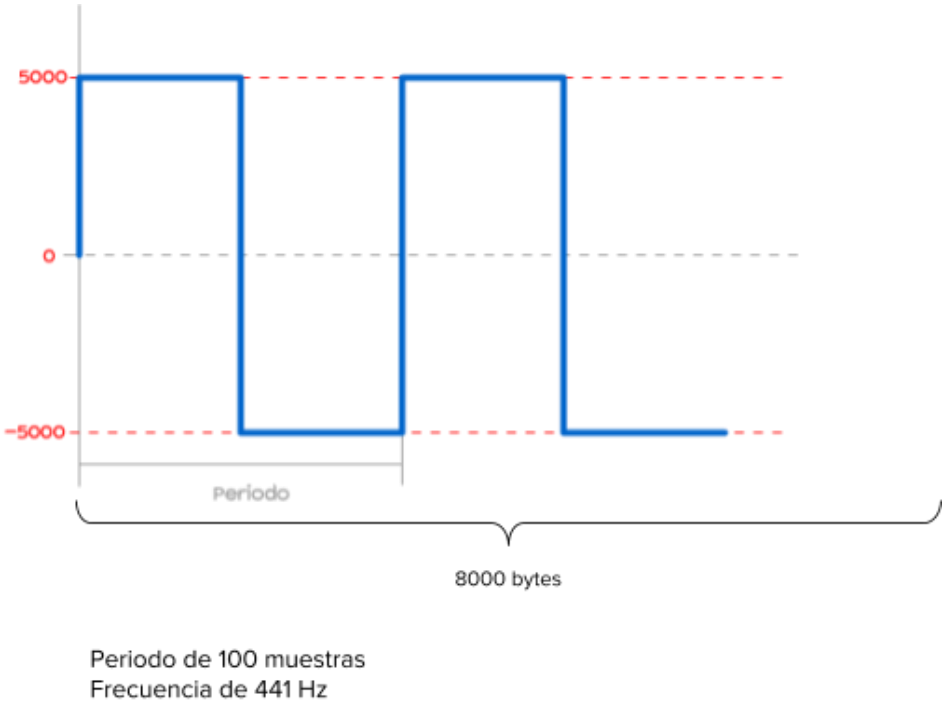


Figura 9.9.3.2: Onda cuadrada

En el caso que nos ocupa, la longitud de audio resultante será el tamaño del paquete definido en la constante *Constante.Configuracion.bufferDatos* que por defecto es de 8000 bytes con una amplitud de 5000 (de un total de 2^{16} por la resolución de la muestra).

10. Aplicación móvil

Una de las principales diferencias entre la versión de escritorio y la de móvil es el enfoque en la interfaz de usuario. Desde el punto de vista de un usuario de un terminal móvil, lo que se busca es, sobre todo una visualización cómoda de la información en pantalla a la hora de interactuar con la aplicación. En una aplicación de escritorio disponemos de ventanas que nos permita cambiar de tarea de una manera muy versátil. En un dispositivo móvil la manera de interactuar es muy distinta dado que normalmente la aplicación no “flota” en la pantalla.

Por otra parte, un punto a tener en cuenta es que una aplicación móvil debe de ofrecer simplicidad en los controles. En una aplicación de escritorio disponemos de hardware que nos permita apuntar, escribir, mostrar funcionalidad con el botón derecho del ratón, hacer *scroll* de la pantalla, etc. En cambio, en la aplicación móvil el hardware del que disponemos es la pantalla táctil y cada vez menos algún que otro botón físico. Por tanto, la situación cambia, y los controles están más enfocados a gestos (*gestures* en inglés); se toca la pantalla, se arrastra el dedo, se amplía mediante dos dedos, etc.

El proyecto en esta plataforma cobra más relevancia puesto que es especialmente útil cuando necesitamos realizar una llamada sin consumir minutos en una llamada clásica.

A pesar de que el diseño de una u otra plataforma es suficientemente distinta como para separar la aplicación en varias distintas, hemos implementado un núcleo en Java que funciona como librería tanto de clases como de funcionalidad. Para ello hemos hecho uso de clases abstractas comentadas anteriormente que permiten exportar atributos y funcionalidad con lo cual simplifica mucho la implementación de la aplicación móvil.

La creación del núcleo de la aplicación para desarrollar una librería ha sido posible dado que el lenguaje en el desarrollo de aplicaciones de Android es Java. Además, dado que existen *frameworks* como Oracle Mobile Application Framework que nos permiten utilizar Java para implementar aplicaciones que funcionan en diferentes plataformas, puede resultar interesante seguir usando la librería para desarrollar la aplicación en otras plataformas.

10.1. Estructura del proyecto

En la siguiente figura vemos que para crear la aplicación en Android, el número de clases necesarias relacionadas con la transmisión y gestión de paquetes así como de datos de aplicación (contactos y registro de llamadas), se simplifica.

Esto es debido a que estamos utilizando el proyecto original de escritorio como librería de clases así como de funcionalidad. Hemos configurado el proyecto de Android para que utilice otro proyecto como dependencia que hemos llamado *GlitchyCore*.

De esta manera, la funcionalidad y las mejoras del núcleo de *Glitchy* se heredan al proyecto de Android. Así, podemos depurar el núcleo del proyecto *Glitchy* para que automáticamente se actualice el proyecto de Android.

Por otra parte, utilizando las clases abstractas desarrolladas, además de simplificar el proyecto, permite implementar la funcionalidad específica de Android.

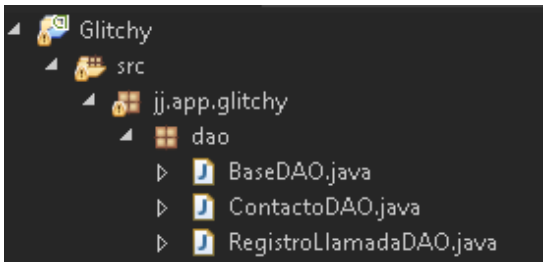


Figura 10.1.1: Clases de la capa de acceso a datos

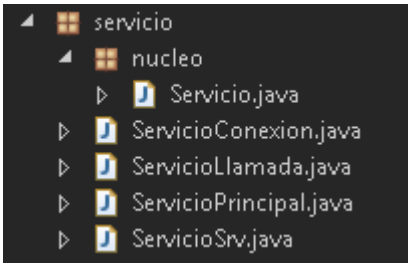


Figura 10.1.2: Clases de la capa de servicio

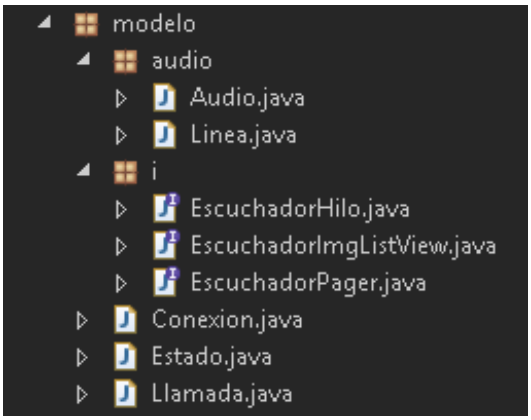


Figura 10.1.3: Clase de la capa modelo

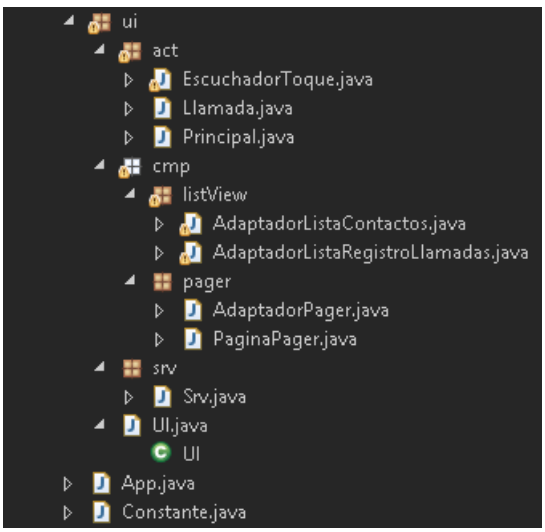


Figura 10.1.4: Clases de la capa de presentación

Con el fin de hacer referencia a las clases del núcleo del proyecto, se ha creado un enlace al proyecto *GlitchyCore*, que contiene el proyecto de la versión de escritorio. De esta manera la aplicación de Android tendrá acceso a las clases abstractas, los modelos, gestión de flujo y funcionalidad.

A continuación, vamos a mostrar una figura en la que se distingue el proyecto de aplicación de móvil de Android llamado *Glitchy* y la aplicación de escritorio *GlitchyCore*.

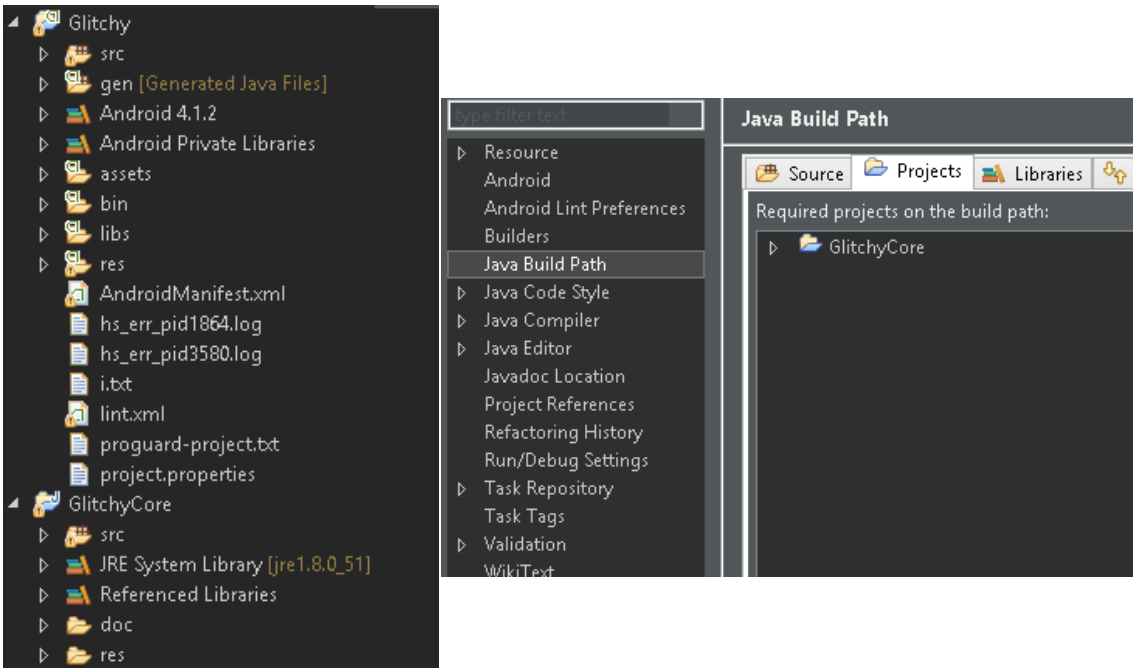


Figura 10.1.5: Relación de la aplicación móvil y la de escritorio (GlitchyCore)

Una vez hemos visto la relación entre los proyectos, vamos a detallar cuál ha sido la estrategia que se ha seguido para implementar las clases de la aplicación móvil.

10.1.1. Capa de acceso a datos

Todas las clases de la capa de acceso a datos (*figura 10.1.1*) heredan de la clase abstracta correspondiente del proyecto *GlitchyCore*.

Así, la clase *BaseDAO* hereda de *CBaseDAO* e implementa los métodos abstractos de ésta, que sirven para acceder al medio de almacenamiento para leer y guardar datos. Recordemos que este acceso es dependiente de la plataforma.

Dado que el funcionamiento de los contactos y del registro de llamadas es igual tanto en la versión móvil como en la versión de escritorio, las clases *ContactoDAO* y *RegistroLlamadaDAO* extienden *CContactoDAO* y *CRegistroLlamadaDAO* respectivamente, heredando, de esta manera, su funcionalidad.

A continuación, en la siguiente figura mostramos que, únicamente heredamos de *CContactoDAO*, no siendo necesaria ninguna implementación adicional. Lo mismo ocurre con la clase *RegistroLlamadaDAO*.

```
public class ContactoDAO extends JJ.app.glitchy.dao.CContactoDAO
{
}
```

```
public class RegistroLlamadaDAO extends JJ.app.glitchy.dao.CRegistroLlamadaDAO
{

}
```

Figura 10.1.1.1: Clase Contacto y RegistroLlamada en la aplicación de Android

10.1.2. Modelo

En cuanto al modelo, la clase *Audio* es una clase que tiene una alta dependencia, pues es la que accede a los dispositivos de reproducción y grabación. Esta clase hereda de la clase *CAudio* implementando los métodos abstractos de la misma.

La clase *Conexion* funciona igual que la de escritorio pero vamos a ampliar su funcionalidad, añadiendo un "escuchador" o *listener* que se ejecuta cuando finaliza el hilo de ejecución. Esta funcionalidad es propia de la aplicación móvil y no ha sido necesaria en la aplicación de escritorio. Por tanto, simplemente heredaremos de la clase *Conexion* de *GlitchyCore*.

Veamos, a continuación, la clase con la funcionalidad ampliada.

```
public class Conexion extends JJ.app.glitchy.modelo.Conexion
{
	EscuchadorHilo escuchador;

	public Conexion(Servicio servicio, EscuchadorHilo escuchador)
	{
		super(servicio);
		this.escuchador = escuchador;
	}

	public void run()
	{
		super.run();
		escuchador.eventoHiloFinalizado();
	}
}
```

Figura 10.1.2.1: Clase Conexion en la aplicación de Android

En cuanto a las clases *Linea*, *Estado* y *Llamada*, ocurre exactamente lo mismo que con la clase *Conexion* anterior, en la que se indica el *listener* a utilizar cuando finalice el hilo.

10.1.3. Servicio

Todos los servicios de la capa de servicio heredan de la clase *Servicio* que se encuentra dentro del paquete *nucleo* de la aplicación de Android como podemos

observar en la *figura 10.1.2*. Dado que esta clase *Servicio* hereda de *CServicio* del proyecto *GlitchyCore*, todos los servicios tienen los atributos y funcionalidad base del núcleo de *Glitchy* que son los mismos que en la versión de escritorio.

10.2. Organización de la aplicación en Android

Como hemos comentado anteriormente, la aplicación en Android difiere de la estructura en una computadora, pues utiliza otra manera de interactuar con el dispositivo y la interfaz.

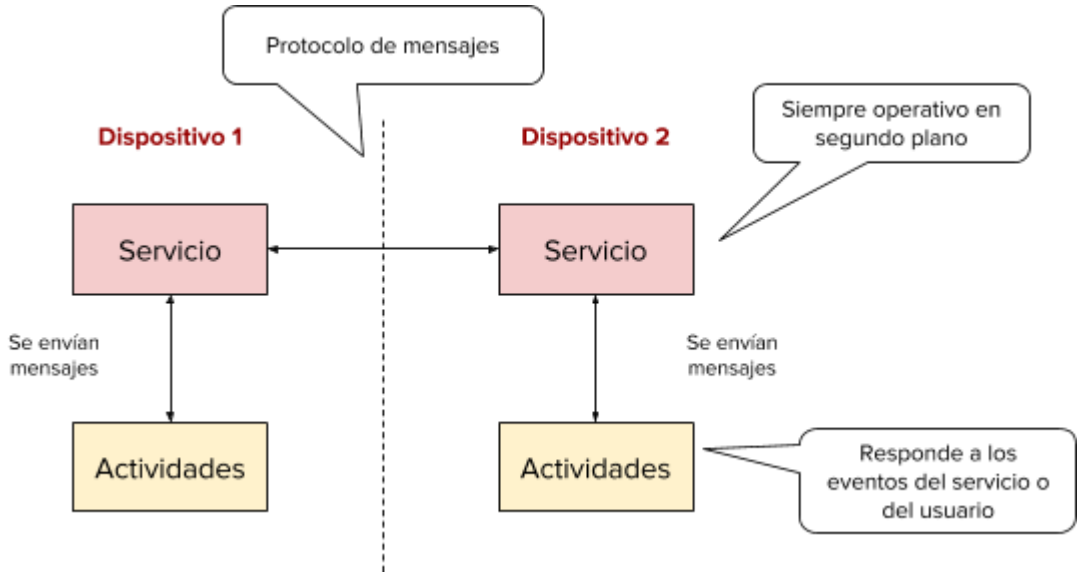


Figura 10.2.1: Esquema básico aplicación móvil

Como vemos, la aplicación está compuesta por un servicio principal y varias actividades con la cual el usuario puede interactuar.

Vamos a detallar cual es el objetivo y las características de cada parte.

- **Actividades**: Las actividades permiten al usuario configurar los parámetros de la aplicación, guardar contactos y realizar llamadas, entre otra funcionalidad.
- **Servicio**: El servicio es el que tiene la capacidad de establecer correctamente la comunicación con el terminal ajeno. Esto se realiza mediante el **protocolo** que hemos definido anteriormente y permite a la aplicación interactuar entre dispositivos. Se entiende con interactuar, la capacidad de establecer comunicación, realizar llamadas, colgar, descolgar, etc.

El servicio y las actividades se **intercambian** unos **mensajes** que comentaremos más adelante para llevar a cabo cierta funcionalidad (llamar, colgar, etc.).

Cada actividad dispone de su propio servicio o controlador en el que se apoya para llevar a cabo la lógica de la misma. No se debe confundir este servicio controlador con

el que se entiende como un servicio en Android (*Service*). El servicio o controlador al que nos referimos es una clase que separa la lógica de la aplicación de la de la interfaz. Un servicio en Android es un componente que puede realizar operaciones de larga duración y se ejecuta en segundo plano.

Esta manera de desarrollar nos permite separar claramente lo que es la parte visual y los eventos de la parte funcional o lógica.

Todos los servicios de cada interfaz o actividad heredan de un servicio principal que dispone de la funcionalidad común a todos.

En la siguiente figura vemos la relación existente entre los servicios y las actividades de la aplicación:

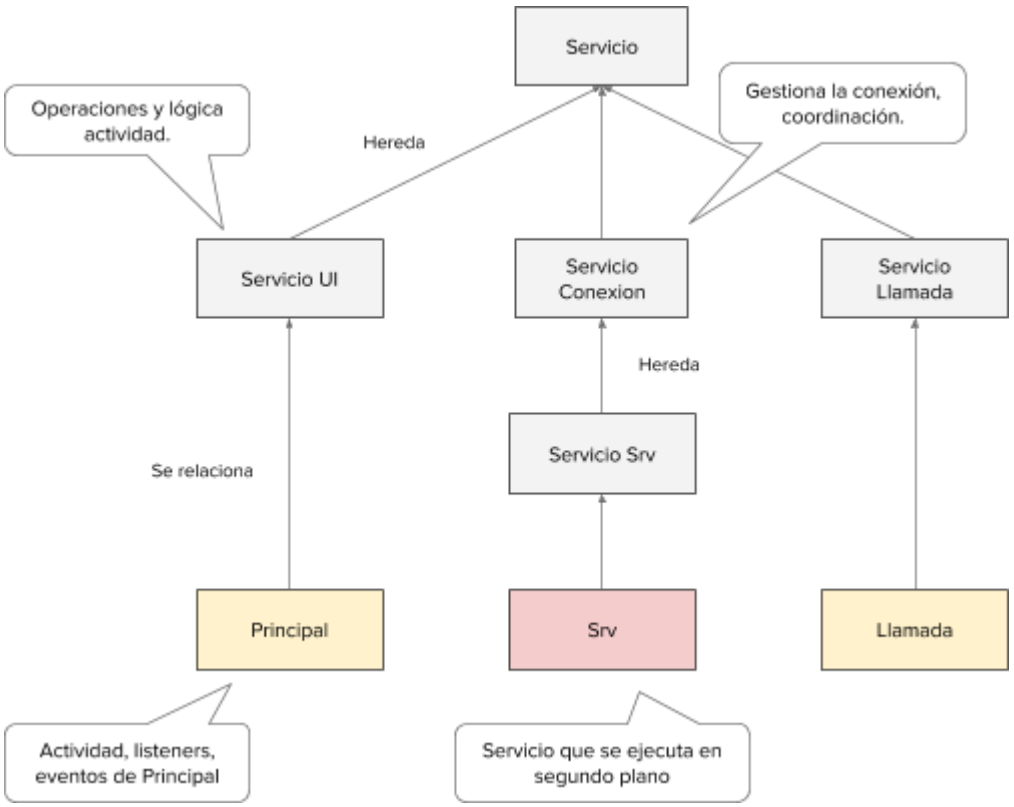


Figura 10.2.2: Relación entre actividades y servicios (controladores)

A continuación listamos las diferentes actividades que conforman la aplicación:

- **Principal**: Esta actividad dispone de la pantalla de configuración, la pantalla de contactos para poder realizar llamadas y la pantalla de *log*.
- **Llamada**: Esta es una actividad volátil, es decir, se lanza cuando se recibe una llamada o procedemos a realizar una. Nos permite colgar y descolgar la llamada.

10.3. Configuración y datos de aplicación

Para poder parametrizar ciertas características de la aplicación, se utilizan ficheros *properties* que son archivos que se componen de diferentes entradas clave, valor.

En cuanto a los datos de aplicación nos estamos refiriendo a la información de los contactos y el registro de llamadas.

Tanto la configuración como los datos de aplicación son almacenados en un directorio de recursos de la aplicación que en Android se denomina *assets*. Este directorio tiene algunas restricciones a la hora de almacenar información ya que los ficheros editables deben de ir en la raíz del mismo. Si disponemos de recursos de solo lectura es posible utilizar subdirectorios. En nuestro caso estos ficheros podrían ser la fuente y la melodía que almacenamos en *res/fonts* y *res/snd* respectivamente.

La estructura que hemos definido en este directorio la mostramos en la siguiente figura:

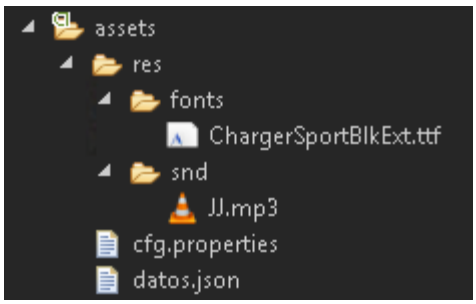


Figura 10.3.1: Estructura directorio assets

10.3.1. Configuración

El archivo de configuración es de tipo *properties* (clave, valor) y nos permite personalizar la configuración del usuario.

El fichero se denomina *cfg.properties* que podemos observar en la *figura 10.3.1* y tiene una estructura interna similar a la siguiente:

```
puerto=20001
tiempoEspera=5000
tiempoEsperaLlamada=20000
melodia=res/snd/JJ.mp3
```

Figura 10.3.1.1: Estructura fichero configuración Android

En la pantalla de configuración con la que el usuario puede interactuar, solo podemos editar los parámetros *puerto* y *melodia*.

10.3.2.Datos de aplicación

Los datos de aplicación, es decir, los contactos y el registro de llamadas tienen una estructura idéntica a la versión de escritorio. Se trata de un JSON que tiene una entrada principal *datos* de la que cuelga *contactos* e *historial* ambos son listas de contactos y registros de llamada respectivamente como podemos observar en la *figura 9.3.1.1*.

10.4. Funcionamiento básico de las actividades

Cuando se inicia la aplicación, se carga el servicio que permanecerá en segundo plano y a la escucha de posibles llamadas entrantes al dispositivo. Para arrancar la aplicación basta con pulsar sobre el icono de la aplicación *Glitchy*.

Figura 10.4.1: Logotipo de Glitchy

10.4.1. Actividad *Principal*

Esta **actividad principal** *Principal*, está compuesta por 3 pantallas. A continuación mostramos un esquema que muestra la composición de pantallas y la comunicación con el servicio.

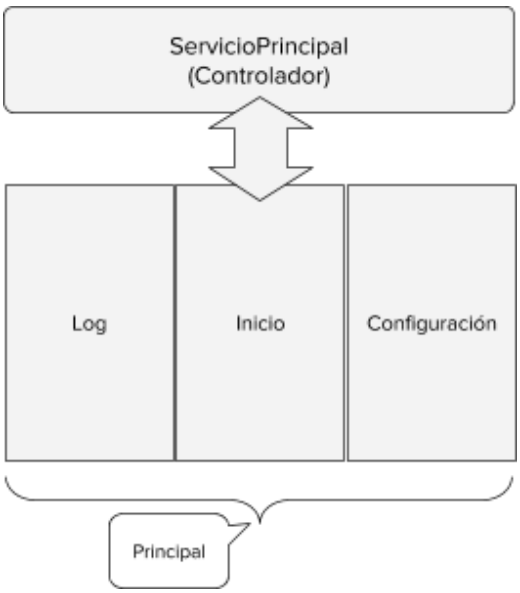


Figura 10.4.1.1: Estructura aplicación

Al cargar la aplicación muestra una pantalla principal en la que se distingue un campo donde insertar la **URI** del destino, un botón para añadir la URI a la lista de contactos almacenados que aparece abajo y un registro de llamadas. En el esquema de la *figura 10.4.1.1* corresponde a la pantalla central (Inicio).

Para definir la manera de interaccionar con la interfaz se ha utilizado el control *ViewPager* que nos permite pasar de una pantalla a otra simplemente deslizando el dedo. A continuación mostramos una imagen de las tres pantallas; la de *log*, la de inicio y la de configuración.

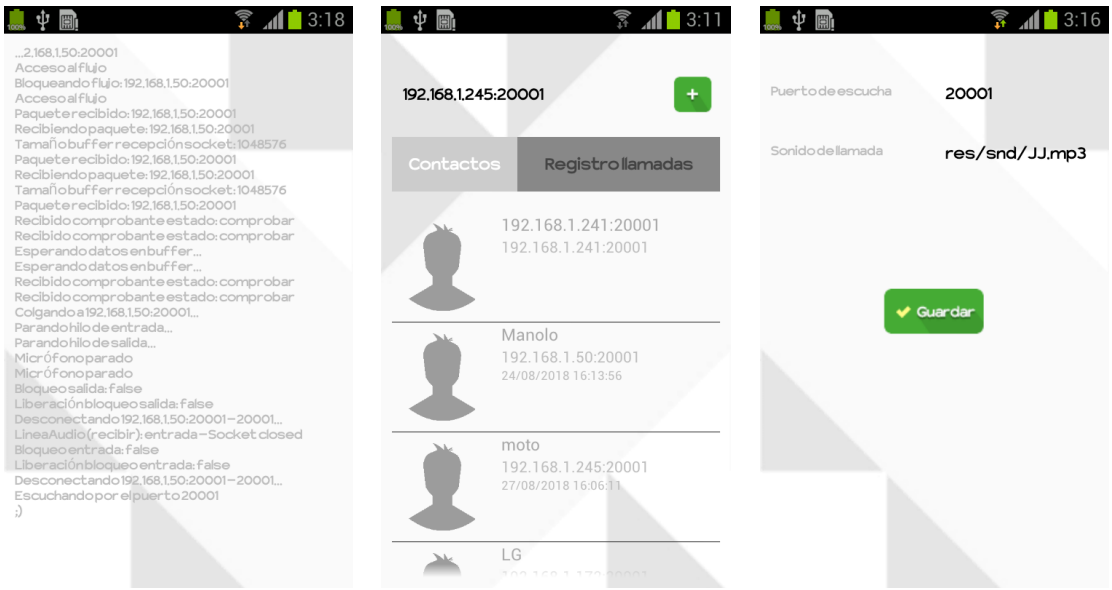


Imagen de la pantalla de log	*Imagen de la pantalla de inicio*	*Imagen pantalla de configuración*

Figura 10.4.1.2: Imágenes de los diferentes pantallazos de la actividad Principal

Al iniciar la aplicación accedemos a la actividad de inicio, de manera predeterminada.

Esta pantalla también dispone de un listado de contactos desplegado por defecto tal y como podemos observar en la siguiente figura (*figura 10.4.1.3*). Además de los controles para introducir la dirección URI y el botón de agregar a contactos, esta pantalla también permite visualizar el historial del registro de llamadas. Esto es posible pulsando sobre el texto “Registro de llamadas” que se encuentra situado a la derecha del de contactos. Seguidamente, mostramos un pantallazo que nos permite visualizar el listado de registro de llamadas.

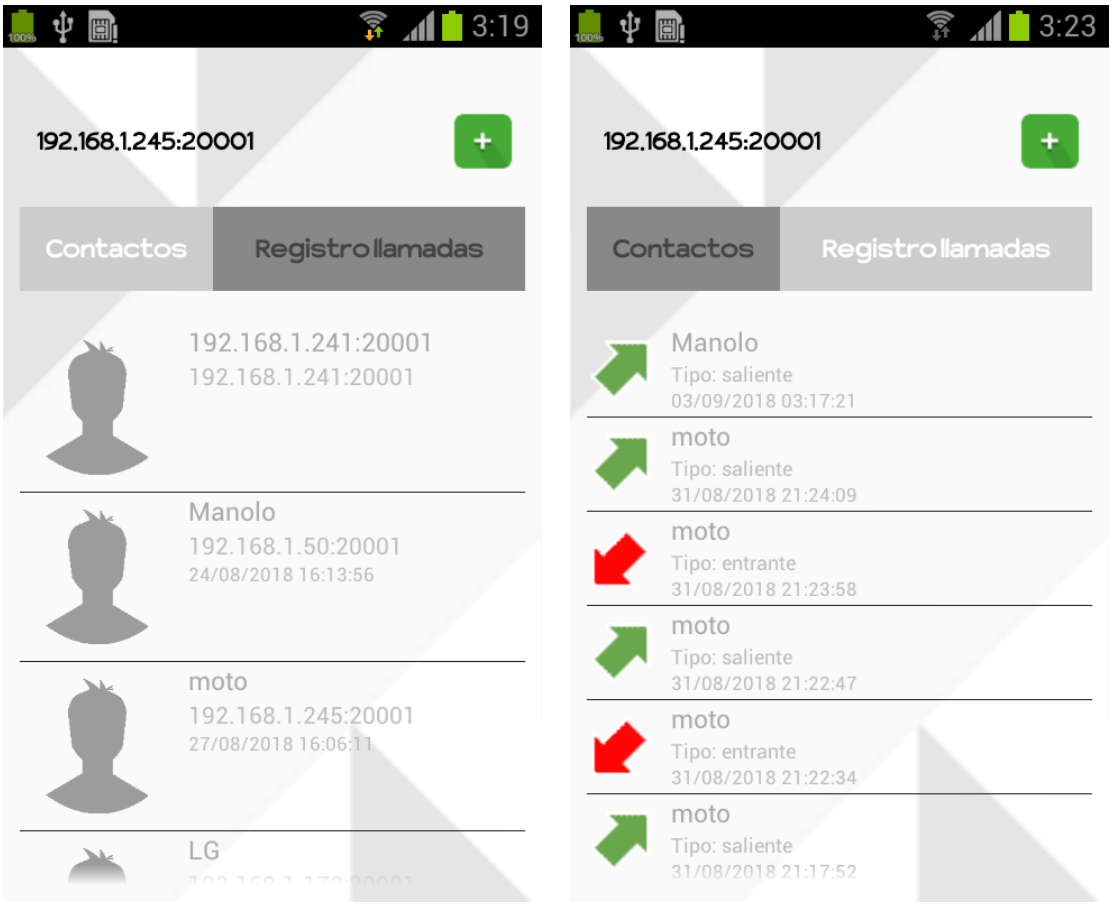


Figura 10.4.1.3: Imagen de la pantalla de inicio mostrando los contactos y el registro de llamadas

Como podemos intuir de la *figura 10.4.1.1*, al deslizar el dedo hacia la derecha desde la pantalla de inicio, accederemos a una pantalla de ***log***, donde se almacenan las trazas o pasos que ha realizado la aplicación durante su ejecución en forma de texto. Esta pantalla está más enfocada a desarrolladores y en el esquema de la *figura 10.4.1.1* corresponde con la pantalla de la izquierda *Log*.

Si desde la pantalla principal de **inicio** deslizamos el dedo hacia la izquierda aparecerá la pantalla de **configuración** de la aplicación donde podemos indicarle el puerto de escucha donde el dispositivo tiene que escuchar para atender las llamadas entrantes. En el esquema de la *figura 10.4.1.1* corresponde con la pantalla de la derecha *Configuración*.

Con el fin de simplificar el diseño de la aplicación, estas tres pantallas están gobernadas por una única actividad *Principal* que mediante un control *ViewPager* permitirá gestionar las pantallas como hemos indicado anteriormente.

10.4.1.1. Gestión de los contactos

Así como ocurre en la aplicación de escritorio, es posible modificar la lista de contactos, agregando o eliminando elementos.

Para agregar un contacto, podemos utilizar el botón con el símbolo más que aparece al lado del cuadro de texto que permite introducir la dirección IP y el puerto del destinatario.

Figura 10.4.1.1.1: Introducción de la dirección destino y botón de agregar a contactos

También es posible **agregar un contacto** realizando una pulsación larga sobre un registro de llamada y seleccionar *Agregar a contactos*.

Para **eliminar un contacto** utilizaremos una pulsación larga sobre el registro del mismo y aparecerá la opción de eliminar entre las diferentes opciones. Pulsando sobre dicha opción, borrará el contacto.

Si deseamos realizar una **llamada a un contacto** existente, basta con realizar el mismo procedimiento que para eliminar un contacto. De entre las opciones que aparecen, pulsaremos sobre *Llamar*.

A continuación mostramos una ilustración con las opciones que aparecen al realizar una pulsación larga sobre un contacto:

Figura 10.4.1.1.2: Opciones que aparecen al realizar una pulsación larga sobre un contacto

Cabe destacar que cuando borramos un contacto, no es posible recuperarlo puesto que se elimina también del medio de almacenamiento.

De forma predeterminada, la imagen de un contacto es el contorno de un usuario pero podemos pulsando sobre la misma. Al hacerlo accederemos al gestor de imágenes por defecto que tenga establecido el dispositivo para seleccionar una imagen y asignarla al contacto.

Este proceso de abrir una aplicación para obtener un resultado es muy utilizado en aplicaciones móviles. A continuación vemos cómo se implementa esta estrategia.

```
Intent seleccionarImg= new Intent(Intent.ACTION_PICK,
android.provider.MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
startActivityForResult(seleccionarImg, n);
```

Figura 10.4.1.1.3: Implementación lanzar actividad para cambiar imagen de contacto

Se trata de crear una petición al sistema, mediante un *Intent* con el parámetro que indique que deseamos seleccionar una imagen en el dispositivo y lanzar la actividad a la espera de un resultado. Si hay varias actividades que cumplan el filtro, aparecerá un cuadro de selección.

La respuesta la obtenemos mediante el método *onActivityResult*, que es llamado automáticamente cuando la actividad recupera el control.

En la siguiente figura podemos analizar el código que permite obtener la respuesta de la actividad.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent imageReturnedIntent)
{
        if (resultCode == RESULT_OK)
        {
                Uri selectedImage = imageReturnedIntent.getData();
                Contacto contacto = (Contacto) lstContactos.getItemAtPosition(requestCode);
                contacto.setRutaImg("" + selectedImage);
                adaptadorContactos.refrescar();
        }
}
```

Figura 10.4.1.1.4: Implementación obtener la respuesta de una actividad

10.4.1.2. Gestión del registro de llamadas

Es posible interactuar sobre el listado de registros de llamadas para borrar uno existente o agregarlo a contactos. Para ello basta con realizar una pulsación larga sobre

el contacto en cuestión. Si el usuario desea utilizar el contenido del registro de llamada para crear un nuevo contacto podrá seleccionar la opción de *Agregar a contactos*. Si el usuario desea eliminar el registro seleccionará la opción *Eliminar*.

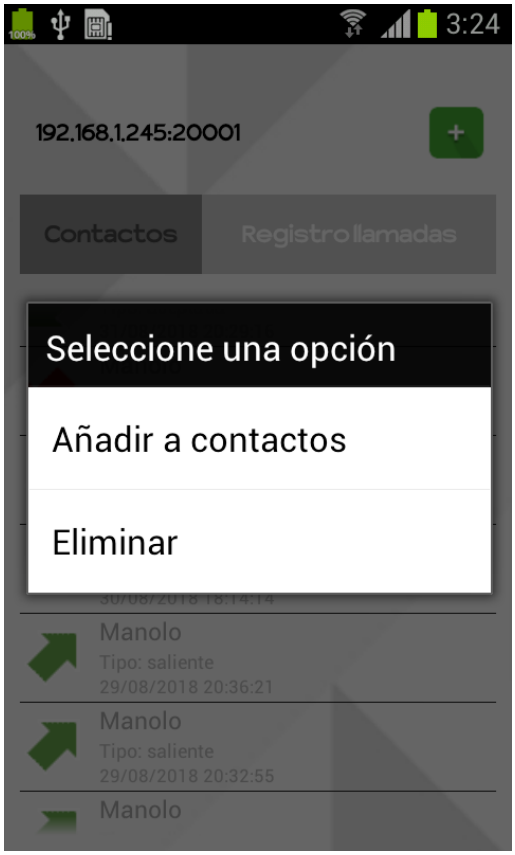


Figura 10.4.1.2.1: Opciones que aparecen al realizar una pulsación larga sobre un registro de llamada

Al igual que ocurre con los contactos, cuando borramos un registro no es posible recuperarlo puesto que se elimina también del medio de almacenamiento.

10.4.1.3. Gestión de la configuración

Desde la interfaz de usuario, en la pantalla de configuración, el usuario podrá modificar algunos de los parámetros de configuración.

En la imagen de la derecha de la *figura 10.4.1.2* podemos observar cómo es la pantalla de configuración de la aplicación móvil. En este caso únicamente es posible modificar el puerto de escucha de llamada y el de transmisión de audio.

10.4.2.Actividad *Llamada*

También existe **otra actividad** *Llamada* que nos permite ser informados de que se está realizando una llamada saliente o nos está llegando una entrante, así como la posibilidad de colgar o descolgar la llamada.

Dependiendo de si se está recibiendo una llamada, se está realizando una saliente o se está entablando una conversación en una llamada en curso, se mostrará una información u otra. Las posibles visualizaciones serán 3 como mostramos en la siguiente figura:

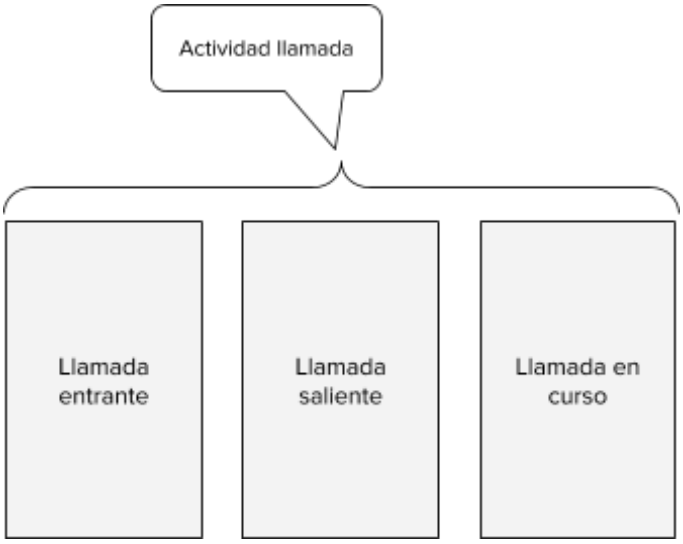
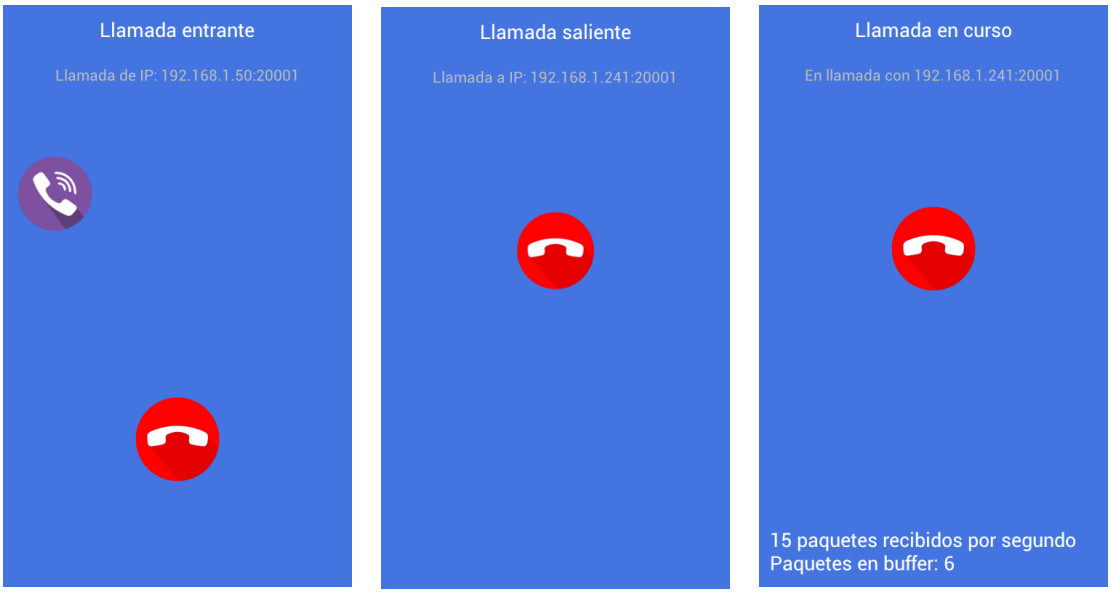


Figura 10.4.2.1: Las tres pantallas de la actividad Llamada



Llamada entrante *Llamada saliente* *Llamada en curso*

Figura 10.4.2.2: Imagen de las diferentes pantallas de la actividad Llamada

10.4.2.1. Llamada entrante

Como hemos comentado, una vez ha arrancada la primera vez la aplicación, el servicio arranca y está a la **espera** de posibles llamadas **independientemente** se cierre o no la aplicación. Esto es lo que diferencia un servicio de Android de una actividad.

Si en algún momento se recibe la petición de llamada por el **puerto** de escucha a la dirección **IP** del dispositivo, el servicio lanzará la actividad *Llamada* con el parámetro *llamadaEntrante* para que ésta muestre la pantalla de llamada entrante. En este momento el usuario tendrá la opción de colgar o descolgar la llamada.

Si el usuario **descuelga** la llamada, se realizará una petición al servicio para que éste realice las operaciones necesarias para abrir el flujo de audio saliente y entrante.

Para descolgar una llamada se deberá deslizar el icono superior de color morado hacia la derecha. Esto se realiza para evitar que se pueda descolgar la llamada sin darnos cuenta.

Si el usuario **cuelga**, se realizará una petición al servicio para que éste realice las operaciones necesarias para terminar la llamada y cerrar correctamente la conexión.

Si el usuario no realiza ninguna acción, saltará el *timeout* o **tiempo máximo de espera** en el servicio que está definido en la configuración de la aplicación. Entonces se cerrará automáticamente la conexión.

10.4.2.2. Llamada saliente

Con el fin de crear una interfaz limpia sin demasiados controles, en la versión móvil de la aplicación, es necesario guardar el contacto previamente para poder realizar una llamada.

Como se ha comentado anteriormente en la gestión de contactos, si el usuario realiza una pulsación larga en cualquier contacto, tendrá la opción de borrar el contacto o realizar una llamada al destino seleccionado (IP y puerto). Esta última acción implica una petición al servicio para que realice la tarea, es decir, realice el protocolo de llamada conectando con el destino en cuyo dispositivo aparecerá la pantalla de llamada entrante.

Desde el dispositivo que realiza la llamada se visualizará la pantalla de llamada saliente y el usuario tendrá, únicamente la **opción de colgar** la llamada. Al igual que en el caso anterior, si no se realiza ninguna acción, el servicio cerrará automáticamente la conexión al saltar el *timeout*, es decir, al transcurrir un tiempo de espera que coincide con el tiempo máximo de espera de la llamada entrante.

Una vez se ha terminado una llamada, el servicio vuelve a permanecer a la escucha para poder recibir nuevas llamadas entrantes.

10.4.2.3. Llamada en curso

Esta pantalla aparece cuando el destino ha aceptado la llamada. En esta situación el usuario tiene la opción de colgar la llamada cuando lo desee. En tal caso, se envía una

petición al servicio de Android para que realice los pasos correspondientes para que se cuelgue la llamada.

Si cualquier usuario sea el que ha realizado la llamada o el que la ha recibido, cuelga, se realiza un procedimiento que provoca que las conexiones del dispositivo local se liberen y finalicen los hilos de ejecución y además, se envía un paquete por la conexión de estado comunicando que el extremo ha colgado la llamada.

En esta actividad se ha implementado una sección para poder visualizar información de depuración y estado.

10.5. Comunicación servicio - actividad

La comunicación entre el servicio y la actividad se realiza tanto mediante mensajes almacenados en la llamada entre las entidades como mediante la emisión de mensajes de difusión local a través de la clase *LocalBroadcastManager*.

Android permite definir *parámetros* al invocar una actividad o servicio que se almacenan como entradas clave-valor.

Para realizar una llamada a una actividad en Android, utilizamos la clase *Intent*. Esta clase nos permite añadir datos *extra* al realizar la llamada a la actividad o el servicio. De esta manera, el servicio o la actividad invocada, podrá obtener los datos adicionales *extra* a través del *Intent* de la actividad o servicio.

En el siguiente ejemplo, tenemos un método que nos permite lanzar una actividad, pasándole por parámetro la clase de la actividad, la acción y un valor:

```
public void lanzarActividad(Class<?> a, String accion, String valor)
{
        Intent i = new Intent(srv, a);
        i.putExtra(Constante.accion, accion);
        i.putExtra(Constante.valor, valor);
        i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        srv.startActivity(i);
}
```

Figura 10.5.1: El método para lanzar una actividad de ServicioSrv en la aplicación de Android

De forma similar, mostramos una manera para enviar un mensaje de la actividad al servicio:

```
Intent i = new Intent(ui, Srv.class);
i.putExtra(Constante.App.accion,
Constante.Comando.Comunicacion.Usuario.descolgar);
i.putExtra(Constante.App.valor, ui.txtIP.getText());
```

```
ui.startService(i);
```

Figura 10.5.2: El método para enviar un mensaje al servicio en la aplicación de Android

Como vemos, para añadir los parámetros *accion* y *valor*, se utiliza la clase *Intent* que es usada después en la invocación a *startActivity* y *startService*.

Hay que tener en cuenta que el funcionamiento de un servicio y una actividad son distintos. Mientras que el servicio permanece cargado desde la primera vez que se ejecuta *startService*, la actividad es una tarea eventual y puede o no estar cargada.

Dado que el servicio y la actividad son procedimientos distintos, es necesario indicar el *flag* *Intent.FLAG_ACTIVITY_NEW_TASK* para cargar la actividad en una nueva tarea si la ésta no está actualmente cargada. Esto presenta un inconveniente al trabajar con hilos, pues si utilizamos este procedimiento para enviar los mensajes de información de paquetes, la actividad Llamada se cargará automáticamente aunque no exista una conexión vigente con el otro extremo. Por tanto, la forma de enviar estos procedimientos, se realiza mediante el envío de mensajes de difusión local mediante los cuales, podemos registrar los receptores de los mismos y definir manejadores para poder interpretarlos. Cuando finaliza la actividad, se cancelará el registro del manejador de recepción de mensajes.

A continuación mostramos la manera de realizar y cancelar el registro, así como la clase anónima que recibe los mensajes:

```
//>> Registrar
LocalBroadcastManager.getInstance(this).registerReceiver(receptorMensajes, new
IntentFilter("evento"));
//>> Cancelar registro
LocalBroadcastManager.getInstance(this).unregisterReceiver(receptorMensajes);
```

Figura 10.5.3: Registro de receptor de mensajes de difusión local

```
private BroadcastReceiver receptorMensajes = new BroadcastReceiver()
{
        @Override
        public void onReceive(Context context, Intent intent)
        {
                servicio.aplicarAccion(intent);
        }
};
```

Figura 10.5.4: Clase anónima del receptor de mensajes de difusión local

Este sistema de mensajes se utiliza únicamente para el envío de mensajes del servicio a la actividad, dado que ésta puede estar cargado o no. Sin embargo, el servicio siempre

está cargado y con el primer método comentado, es suficiente para comunicarnos con él.

10.5.1. Acciones

El listado de acciones que se pueden enviar al servicio desde las actividades son las siguientes:

1. **iniciar**: La actividad envía el comando de iniciar al servicio para que éste simplemente arranque.
2. **llamadaSaliente**: La actividad *Llamada* envía la petición de llamada al servicio para que éste conecte y realice la llamada.
3. **colgar**: La actividad *Llamada* le comunica al servicio que desconecte la conexión y éste se encargará de eliminar la pantalla de llamada.
4. **descolgar**: La actividad *Llamada* le comunica al servicio que lance los hilos de audio y entonces éste mostrará la pantalla de llamada en **conversación**.

El listado de acciones que se pueden **enviar a la actividad desde el servicio** son las siguientes:

1. **llamadaEntrante**: El servicio lanza la actividad *Llamada* para que muestre la pantalla de llamada entrante.
2. **llamadaSaliente**: El servicio envía la petición a la actividad *Llamada* para que muestre la pantalla de llamada saliente (después de haber conectado con el destino).
3. **llamada**: El servicio lanza la actividad *Llamada* para que muestre la pantalla de llamada en modo **conversación**.
4. **finalizar**: El servicio le comunica a la actividad *Llamada* que finalice.
5. **debug**: El servicio puede enviar información de depuración a la actividad *Llamada* para que muestre información en pantalla.

11. Pruebas

Las pruebas que hemos realizado consisten en testear la aplicación en diferentes plataformas, así como en diferentes entornos de red.

En primer lugar, hemos configurado un contexto base de pruebas para poder llevarlas a cabo de forma equitativa en cuanto a la configuración de la aplicación se refiere.

La información de la tabla que, a continuación mostramos, se ha establecido tanto en la aplicación móvil como en la de escritorio.

Características de configuración inicial:

Tamaño buffer de datos (bytes)	8000
Tamaño buffer de contención (paquetes)	8
Número de paquetes para calcular paridad	6
Umbral de sonido máximo para silencio	1000 (de 32768)

Tabla 11.1: Configuración de pruebas inicial

El tamaño del buffer de datos es el tamaño del paquete de aplicación dedicado a los datos de audio a transmitir. En la *figura 6.1.1* podemos ver el formato del paquete.

El tamaño del buffer de contención es el tamaño en número de paquetes del buffer de entrada.

El número de paquetes para calcular la paridad, significa cada cuántos paquetes uno debe ser de paridad.

El umbral de sonido máximo indica hasta qué valor de muestra o de amplitud de onda dentro de los datos de audio de un mismo paquete se puede considerar silencio.

11.1. Prueba 1: De PC a terminal móvil en LAN

Se desea realizar una llamada a un PC de escritorio desde un terminal móvil en una red local con Wi-Fi. A continuación mostramos una serie de tablas con la configuración de los recursos que intervienen.

Características de la red:

Tamaño de la red	LAN
Tipo de red	Wi-Fi 802.n y cable par trenzado CAT 5.

Tabla 11.1.1: Características de la red de la prueba 1

Características de los equipos:

Equipo	PC	Terminal móvil
Sistema operativo	Windows	Android
Interfaz de red	cableada	Wi-Fi

Tabla 11.1.2: Características de los equipos de la prueba 1

En esta prueba, hemos observado que el funcionamiento es correcto, la calidad de audio es muy clara dado que estamos utilizando 16 bits a 44100Hz. Observamos que el buffer se conserva entre 2 y 5 elementos. Como se ha comentado anteriormente, el hecho de que el buffer sufra fluctuaciones se puede considerar normal, dado que a medida que se reproduce el audio, se van consumiendo paquetes y el hecho de almacenar los datos del micrófono en la máquina remota y enviarlos a la red, para que finalmente se agregue al buffer de la máquina local, supone un tiempo que puede diferir bastante que el de reproducción.

Debemos, además, tener en cuenta que cuando generamos el paquete de paridad, se envían 2 paquetes consecutivos. El interés principal es que el buffer permanezca con registros disponibles.

En cuanto al buffer de salida, el funcionamiento es diferente, pues, el mismo se va rellenando, hasta calcular el de paridad, en cuyo instante se vaciará para comenzar de nuevo, tal y como se ha comentado en el capítulo “Buffer de salida”.

En la siguiente ilustración, podemos observar el número de paquetes que entran en el buffer de entrada y salen por el de salida que, en el instante de la captura, coinciden.

Cabe señalar el hecho de que los paquetes en el buffer de entrada aparecen como **disponibles**, pues todavía no se han reproducido. Sin embargo, el significado del campo disponible en los paquetes de salida, se refiere a que estos paquetes ya han sido enviados. Es por ello que aparecen como no disponibles. En la sección de “7.1 Buffer de entrada” y “7.2 Buffer de salida” se comenta el funcionamiento de este atributo.

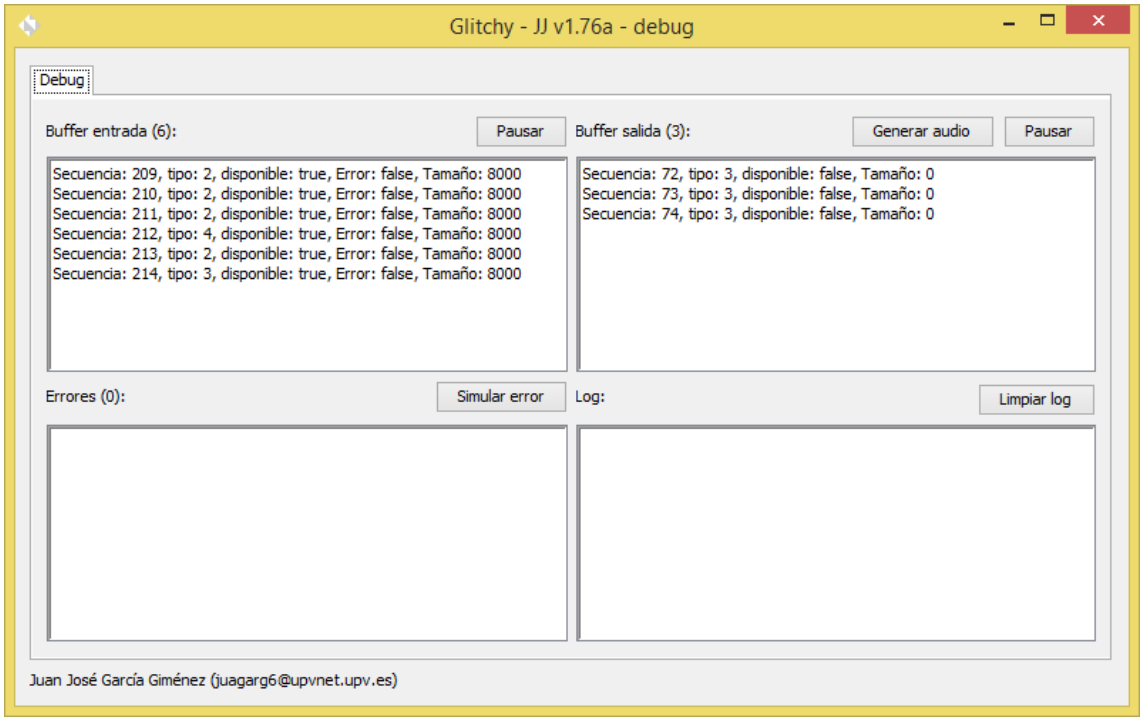


Figura 11.1.1: Paquetes enviados y recibidos en un instante de la prueba 1

No se han observado, errores en la comunicación.

11.2. Prueba 2: De terminal móvil a terminal móvil en LAN

Esta prueba ha resultado más compleja de gestionar, pues no disponemos de una pantalla en la que visualizar claramente la información de los paquetes, pero nos resultará de gran ayuda el apartado en el que se muestra información sobre los paquetes que se envían, los que se reciben y los mensajes de estado de la comunicación en la Actividad *Llamada*.

Características de la red:

Tamaño de la red	LAN
Tipo de red	Wi-Fi 802.11 b/g/n

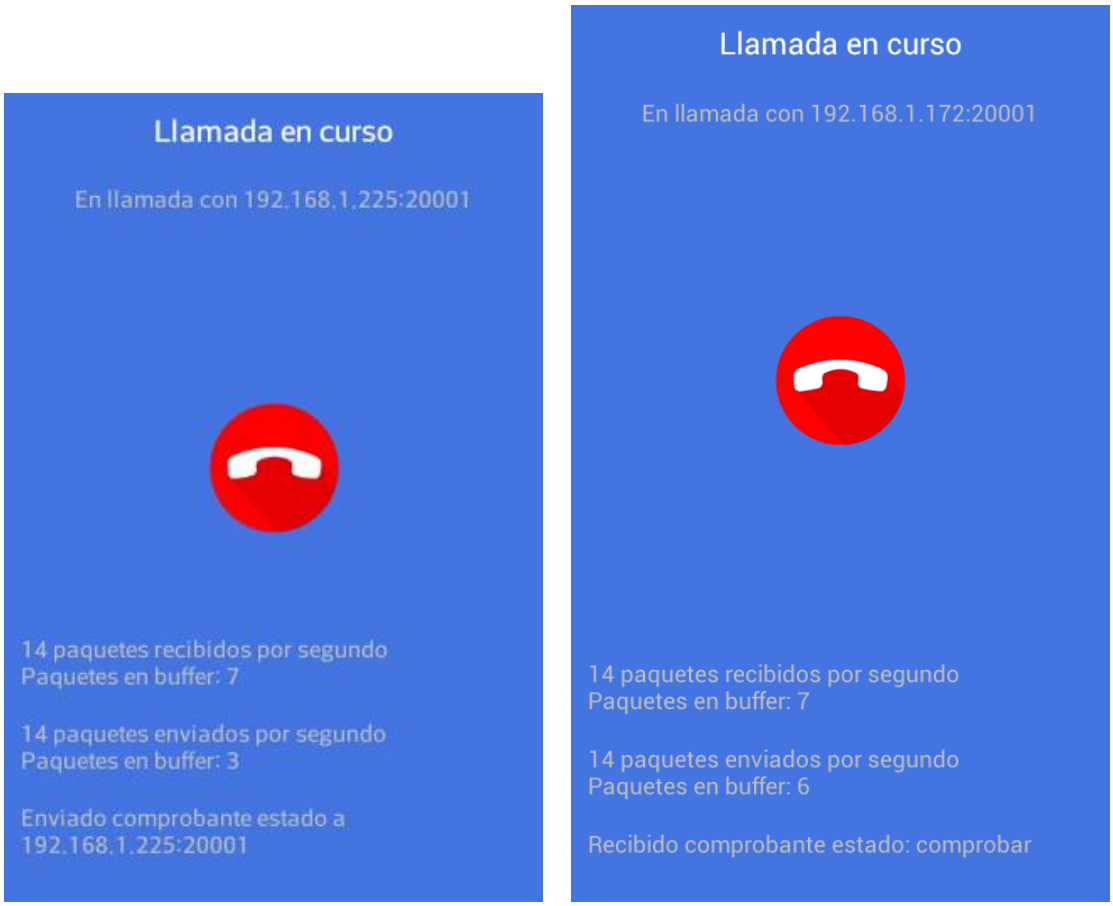
Tabla 11.2.1: Características de la red de la prueba 2

Características de los equipos:

Las pruebas se han realizado con un terminal LG Optimus L5 y un Samsung Galaxy S Duos S7562, ambos de 4 pulgadas.

Equipo	LG Optimus L5	Samsung Galaxy S Duos S7562
Sistema operativo	Android 4.0.3	Android 4.0.4
Interfaz de red	Wi-Fi 802.11 b/g/n	Wi-Fi 802.11 b/g/n

Tabla 11.2.1: Características de los dispositivos de la prueba 2



LG Optimus L5 *Samsung Galaxy S Duos S7562*

Figura 11.2.1: Dispositivos de la prueba de comunicación de dos terminales móviles

En este caso vemos que el número de paquetes enviados por segundo son de 14 que coincide con los paquetes recibidos por segundo. Este resultado es el esperado dada la configuración que hemos indicado anteriormente. Vamos a desarrollar matemáticamente la explicación de la afirmación anterior:

Por una parte tenemos que el formato de reproducción y lectura de datos de audio digital es de 44100Hz a 16bit/muestra.

Por otra parte, hemos configurado la aplicación para que la comunicación se lleve a cabo transmitiendo paquetes de 8002 bytes. Dado que 2 bytes son para indicar el número de secuencia y el tipo de paquete, nos quedan 8000 bytes de datos de audio.

Cuando indicamos que el formato de audio es de 16bit a 44100Hz, estamos diciendo que se van a procesar 44100 muestras por segundo y además, cada muestra tiene un tamaño de 16 bit que corresponden a 2 bytes.

Por tanto, con una sencilla división podemos obtener el número de paquetes de datos por segundo que se enviarán teniendo en cuenta el tamaño del mismo.

$$\frac{44100 \times 2}{8000} = 11{,}025\ paquetes$$

El resultado es algo más de 11 paquetes, como estamos realizando las operación en base a paquetes completos, tendríamos 12 paquetes completos. Pero a esta cantidad hay que sumarle los paquetes de paridad que también se envían. En la tabla de configuración de la aplicación vemos que de cada 6 paquetes, uno es de paridad, con lo que se genera uno de paridad cada 5 paquetes. Por tanto, a los 12 paquetes hay que sumarle 2 de paridad, obteniendo así los 14 que nos aparece en las imágenes de la figura.

El número de paquetes en el buffer de entrada, rara vez sobrepasará el número máximo definido en las constantes, pues en el momento que llegue al máximo, empezarán a ser reproducidos o despachados por ser de paridad (información XOR no reproducible).

Al igual que en la prueba anterior, el funcionamiento ha sido satisfactorio en las diferentes llamadas realizadas.

11.3. Prueba 3: De PC a PC a través de internet

Cuando realizamos ensayos en nuestra red LAN, podemos garantizar cierta estabilidad y seguridad en la comunicación dado que la información que se transmite está dentro de nuestro control y por tanto, tenemos una gestión más exhaustiva sobre el ancho de banda.

Sin embargo, fuera de la LAN, nos encontramos en un entorno en el que no tenemos ese control y necesitamos abrir los puertos de nuestros *routers* para que sea posible el acceso remoto. Hemos tenido que configurar el *router* de acceso a internet de cada extremo para que nos podamos comunicar a través de internet.

Como ya sabemos, el puerto por defecto que utiliza la aplicación es el 20001. Se utiliza la misma conexión para para transmitir información de control y estado como para el flujo de datos de audio.

Dado que estamos comunicándonos a través de internet, podrían existir retardos o congestión en la red a causa de rutas largas o colas en los *routers*. En esta situación, la

implementación de las estrategias de control de congestión pasa a un plano más relevante.

Con el fin de obtener resultados diversos y representativos, hemos modificado los parámetros para que la comunicación sea más estable:

Tamaño buffer de contención (paquetes)	12
Número de paquetes para calcular paridad	4

Tabla 11.3.1: Modificaciones configuración de la prueba 3

De esta manera, tenemos más buffer para almacenar paquetes y enviamos una tasa de paquetes de paridad más alta con el fin de que la aplicación disponga de más información para recuperar los paquetes perdidos. Si bien es cierto, ampliar el tamaño del buffer, aumenta el retardo en el envío de los paquetes pero también disminuye los posibles cortes ante retardos.

Para realizar esta prueba nos hemos conectado por escritorio remoto a una máquina remota para poder visualizar la pantalla de depuración en la misma.

Vamos, a continuación, a analizar los datos que se transmiten los extremos en un instante determinado.

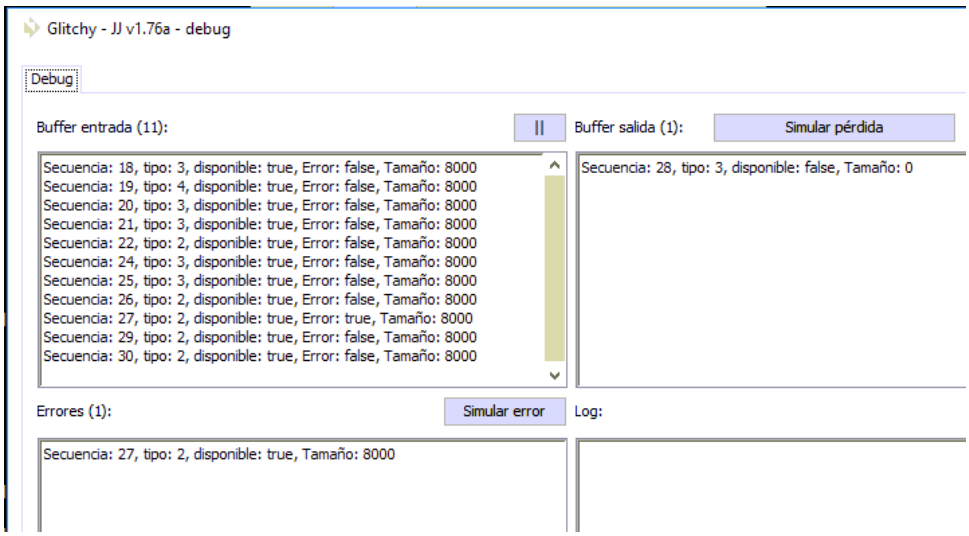


Figura 11.3.1: Imagen de la ventana de depuración de la máquina remota

Dado que el tamaño de los paquetes es mayor que la unidad máxima de transferencia o MTU (*Maximum Transmission Unit*) de Ethernet (1500 bytes), hemos observado que se produce el fenómeno de fragmentación, lo que nos ha provocado algunas incidencias a la hora de la reconstrucción de paquetes. Debemos tener en cuenta que si se pierde un fragmento, se pierde todo el datagrama.

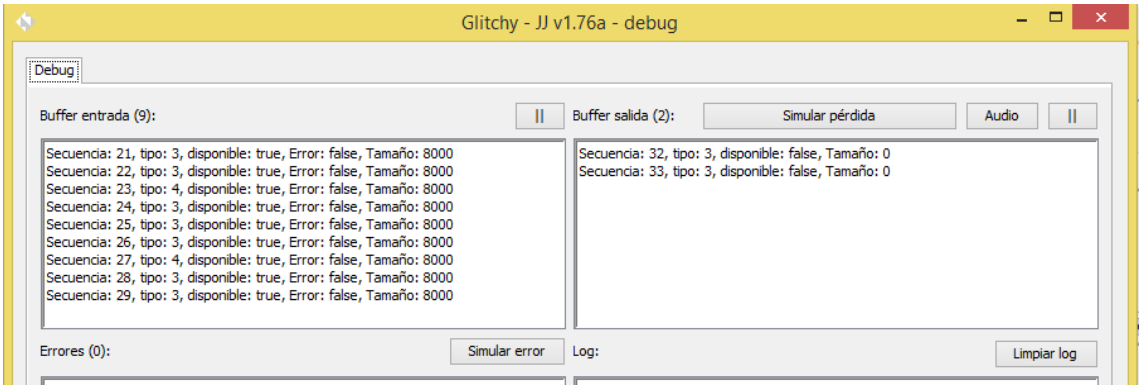


Figura 11.3.2: Imagen de la ventana de depuración de la máquina local

En las figuras anteriores podemos observar los paquetes recibidos y enviados tanto en la máquina remota como en la máquina local en un instante ***t***.

Debido a que la pantalla de depuración que observamos en las figuras tiene un periodo de refresco de 250 ms, los datos mostrados en este instante ***t***, pueden tener un error de 250 ms, que debemos tener en cuenta en la lectura de resultado. A esto se puede sumar el retardo de refresco de la conexión realizada mediante escritorio remoto.

En esta ocasión, hemos experimentado, pérdida de paquetes en la máquina remota. Sin embargo, no se han encontrado incidencias en la máquina local. Cabe indicar que únicamente, hemos enviado información de audio desde la máquina local.

Como observamos en la *figura 11.3.1*, el paquete 27 no se ha recibido, lo que provoca que nuestro algoritmo, cree una entrada en el buffer de entrada con el campo de error a *true*, con el fin de que posteriormente intente recuperarlo cuando obtenga el paquete de paridad.

Cuando recuperamos un paquete añadimos una marca para conocer su estado. En la figura siguiente, observamos como el paquete con número de secuencia 41, ha sido recuperado correctamente, antes de ser reproducido. Esta recuperación se ha llevado a cabo a partir del paquete 43 de tipo paridad que ha recibido la aplicación.

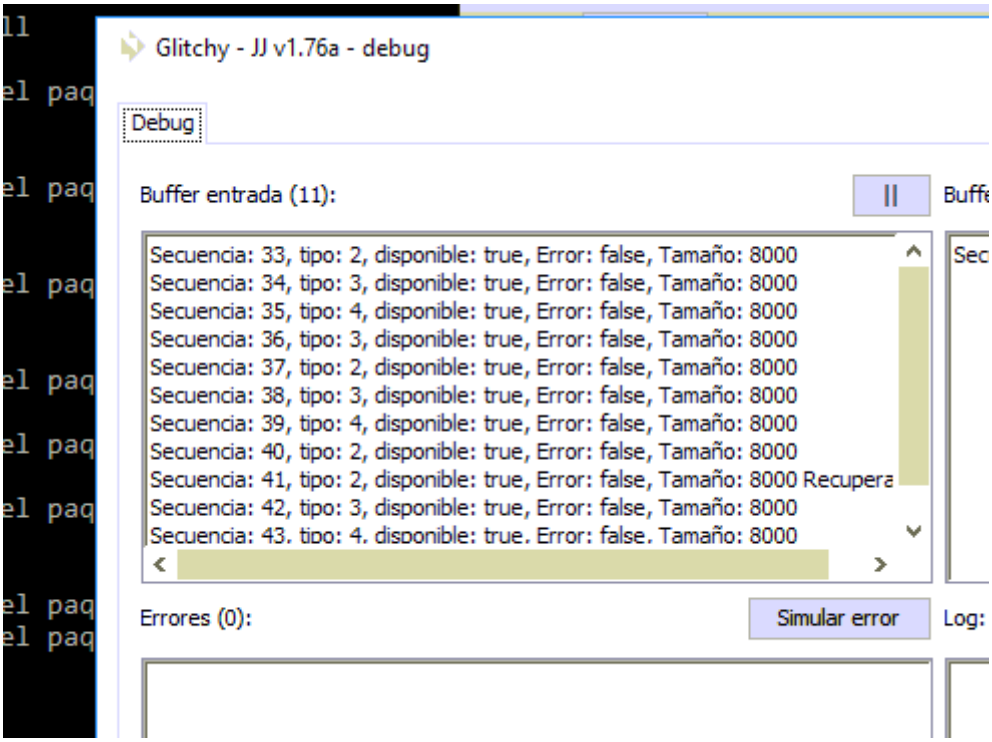


Figura 11.3.3: Imagen mostrando un paquete recuperado en la máquina remota

En cuanto a los paquetes 32 y 33 enviados desde la máquina local (*figura 11.3.2*), podemos deducir que, en este momento, o bien, están de camino, o bien, han llegado antes del último refresco de la ventana de la máquina remota (*figura 11.3.1*).

11.4. Prueba 4: De terminal móvil a PC a través de internet

Como hemos comentado en los límites del proyecto, existen limitaciones a la hora de acceder a un puerto de un terminal móvil a través de la red de datos debido a que en la conexión puede existir una NAT, por tanto, lo que hacemos es realizar una llamada **desde un terminal móvil** a un PC con línea fija. De esta manera, se utiliza el puerto efímero para comunicarse con el terminal.

La siguiente figura ha sido obtenida mediante el programa PingTools. Como podemos observar, existe una interfaz entre nuestro dispositivo y la IP pública.

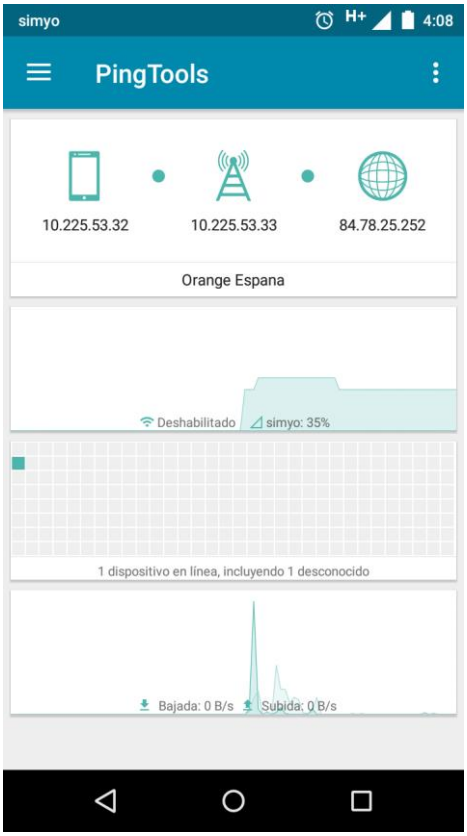


Figura 11.4.1: Imagen de las características de la conexión de datos del terminal

Con la configuración utilizada en las pruebas anteriores, experimentamos un retardo que quizás no es lo suficientemente bajo como para considerar una interacción en tiempo real. Podemos considerar que una aplicación es interactiva cuando su retardo es inferior o igual a 0,4 segundos.

En este caso hemos ajustado los parámetros para intentar conseguir que el retardo no sobre pase los 400 ms.

Se han realizado las pruebas con los parámetros siguientes:

Tamaño buffer de datos (bytes)	1400
Tamaño buffer de contención (paquetes)	20
Número de paquetes para calcular paridad	10

Tabla 11.4.1: Modificación configuración de la prueba 4

Como vemos en la tabla anterior, también conviene ajustar los parámetros del buffer de contención y el número de paquetes para calcular la paridad. Esto es necesario dado que con un tamaño del buffer más pequeño, la velocidad con que se despachan los paquetes será más alta y el buffer se vaciará más rápido.

Con esta configuración, además de evitar la fragmentación (1400 < MTU Ethernet), conseguimos tiempos de hasta 320 ms. A continuación, veamos cómo hemos calculado este valor.

Si partimos de que se está codificando la información de audio a (44100 x 2) bytes por segundo, deducimos que 1400 bytes tardarán en codificarse unos 16 ms. Este valor lo multiplicamos por 20 que es el número de paquetes a introducir en el buffer de entrada antes de empezar a reproducirlos, dándonos como resultado los 320 ms.

Al valor de 320 ms calculado anteriormente, hay que sumarle, además, el tiempo que tarda el paquete en llegar a su destino.

En la figura que mostramos a continuación, podemos ver el tiempo que tarda un paquete en ir y volver desde el terminal móvil al PC de escritorio.

Figura 11.4.2: Tiempos de latencias entre el terminal móvil y el PC

Sumando los 320 ms con la media de los tiempos de propagación, obtendríamos aproximadamente 350 ms en condiciones óptimas, valor que consideramos aceptable para el cometido del proyecto. No obstante, en la práctica, estos valores pueden fluctuar mucho en cualquier momento debido a la naturaleza inalámbrica de la conexión móvil, pudiendo superar el umbral de interactividad de 400 ms.

Presentamos, a continuación, la imagen del terminal móvil una vez realizada la llamada.

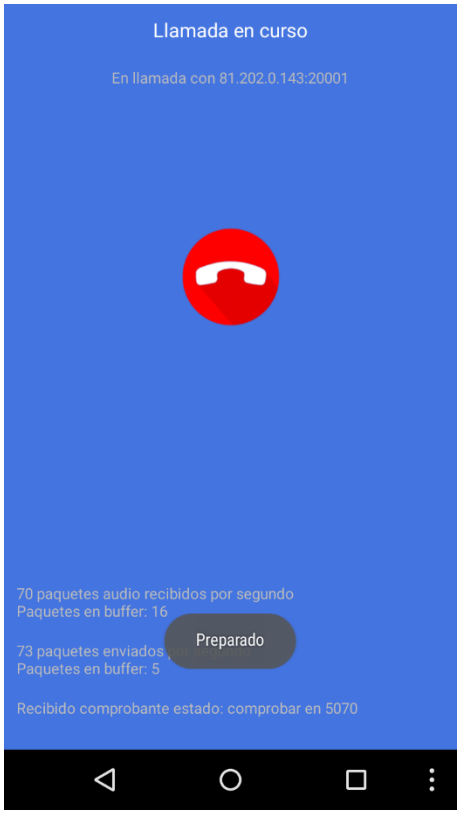


Figura 11.4.3: Imagen del terminal móvil remoto

Observamos que las cifras son bastante superiores en cuanto a la tasa de envío y el buffer de entrada permanece entre 14 y 18 paquetes, aunque hemos experimentado fluctuaciones considerables en situaciones puntuales.

Veamos, a continuación, la pantalla de la máquina local.

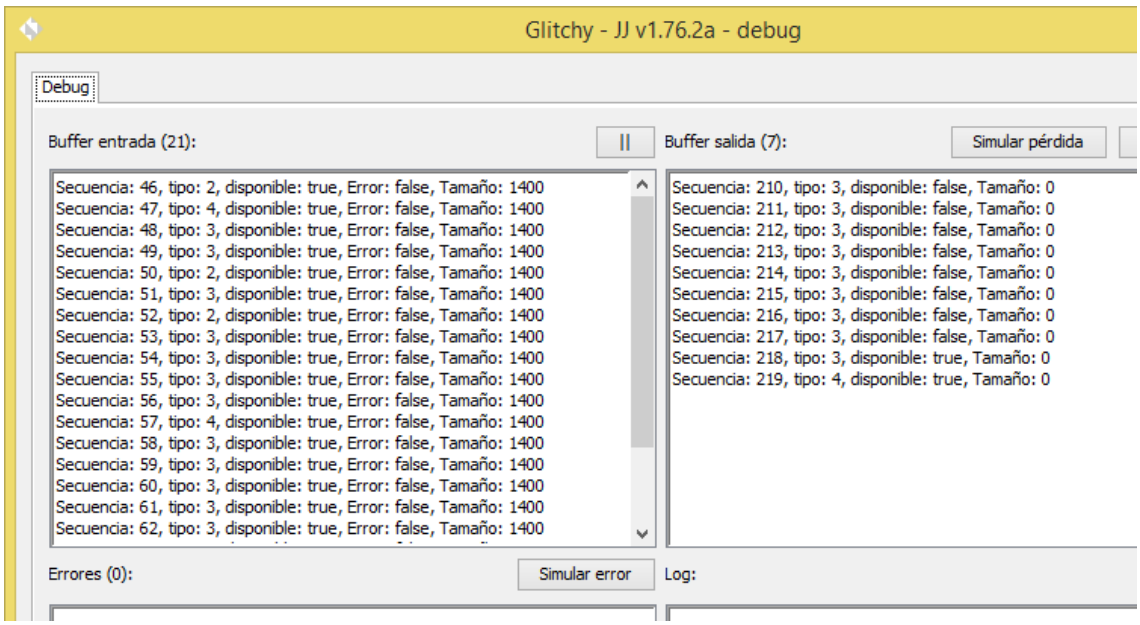


Figura 11.4.1: Información de los paquetes de la prueba 4

Observamos en el buffer de entrada, cómo estamos transmitiendo voz desde el terminal móvil hacia la máquina local (paquetes de tipo 2) junto con silencios y paridad. En la máquina local no aparece ningún paquete con datos de audio en el buffer de salida dado que no se está hablando por el micrófono y el sonido ambiental se elimina. Es por esto último, que los paquetes enviados desde la máquina local tienen un tamaño de datos de audio de 0 bytes.

También podemos observar cómo se ha generado el paquete de **paridad** (tipo 4) en la máquina local justo después del noveno paquete, tal y como se indica en la configuración de la aplicación.

Los valores presentados en esta prueba son los resultados esperados y ha sido posible entablar conversación con una calidad suficientemente buena.

12. Conclusiones

12.1. Trabajo realizado

A la hora de realizar este proyecto, hemos llevado a cabo una serie de pasos con el fin de abordarlo de la forma más coherente posible.

Lo más importante de este proyecto ha sido la motivación y el reto que supone poder realizar una comunicación en tiempo real desde la raíz.

En primer lugar, hemos hecho un análisis para describir otras soluciones similares existentes, su funcionamiento, limitaciones y sus protocolos para conocer dónde estamos y a dónde queremos llegar.

A continuación, se ha planteado una funcionalidad general de lo que debe de hacer la aplicación en un análisis de objetivos para seguidamente, identificar y modelar las partes que componen la aplicación, así como detallar cuáles son los límites y necesidades de implementación.

Profundizando en la materia, hemos repasado algunos conceptos generales del audio digital y su codificación y así continuar con el diseño de algunos protocolos para encapsular la información a transmitir en paquetes de aplicación. Esto nos ha llevado a definir unas estrategias de resolución de problemas que pueden ocurrir en la red dado que nuestra aplicación funciona a través de la misma.

Una vez realizado el diseño general, hemos procedido con la descripción de la arquitectura del proyecto y la implementación tanto para la versión de móvil como la de escritorio.

Terminada la implementación, damos pie a un capítulo de pruebas que, a pesar de los problemas encontrados, hemos realizado de forma satisfactoria.

12.2. Problemas encontrados

Uno de los primeros problemas encontrados y no por ello menos importante, es el de realizar una aplicación que debe funcionar a través de la red mediante un protocolo propio, además, debe funcionar tanto como cliente y como servidor. Esto nos ha complicado en cierta medida la implementación inicial de la aplicación completa dado que para desarrollar una parte, necesitamos la otra y para ello hemos comenzado por implementar un tanto de improviso la aplicación. En esta situación, el análisis de la situación ha sido fundamental. Eclipse nos ha sido de gran ayuda para llevar a cabo la tarea, dado que es portable y nos permite ejecutar varias instancias de la aplicación. Sin embargo, para un desarrollo en un contexto más realista, ha sido necesario utilizar varias máquinas.

Otro punto especialmente crítico, han sido los hilos de ejecución. Trabajar con hilos de ejecución implica un desarrollo y depuración un tanto exquisita, pues debemos implementar estrategias de bloqueo a recursos para evitar problemas de acceso concurrente.

Normalmente, el hecho de depurar una aplicación cuando se hacen varias cosas a la vez, no representa necesariamente un contexto realista y, por tanto, no ha sido una tarea sencilla y puede resultar más complicado encontrar el problema que solucionarlo. Por ejemplo, tal y como se ha comentado en el apartado de "8.2 Consideraciones de implementación", se han tenido que implementar métodos bloqueantes con el fin de garantizar el acceso al buffer de forma sincronizada.

Por otra parte, al realizar las pruebas, hemos observado problemas de rendimiento en algunos terminales lo que nos ha llevado un tiempo detectar. Mostrar el número de paquetes por segundo tanto los que se envían como los que se reciben nos ha permitido llegar a esta conclusión. Por tanto, han ocurrido incidencias de vaciado del buffer y, por ende, cortes en la comunicación con algunos terminales. Esto ocurre cuando el dispositivo está vinculado con el entorno de desarrollo, funcionando correctamente en caso contrario.

Debido a que el tamaño mínimo de datos aceptado para muestrear el audio del micrófono en los terminales de Android con los que hemos desarrollado la aplicación, se encuentra en 4096, hemos tenido la necesidad de implementar una estrategia para poder disminuir el tamaño mínimo de datos a almacenar en el paquete de aplicación. Esta estrategia consiste en dividir los datos leídos en paquetes más pequeños para posteriormente, enviarlos a la red de forma independiente.

No podemos evitar mencionar que hemos tenido incidencias a la hora de testear el programa con ciertas máquinas como el PC remoto de la prueba 1 "Prueba 3: De PC a PC a través de internet" dado que se perdían paquetes que, a priori, deberían de recibirse. Después de hacer todas las pruebas oportunas, hemos observado que si evitamos la fragmentación, la comunicación tiene más garantías de llevarse a cabo con mayor fluidez.

12.3. Valoración

Hemos repasado el funcionamiento del audio digital y cuáles han sido las estrategias para trabajarlo. Esto nos permite resolver muchas dudas en cuanto a su interpretación digital y nos da las bases para implementar una manera de establecer comunicación y poder entablar conversación con otra persona que se encuentre en cualquier parte del mundo con conexión a internet. Esto nos ha servido como punto de partida para conocer cuáles son los pasos que podemos seguir para transmitir información a través de la red.

En el desarrollo del proyecto, nos hemos dado cuenta del por qué y la importancia de la implementación de protocolos. Hemos podido comprobar cómo poner en práctica la base teórica adquirida.

Nos hemos enfrentado a un gran reto que no ha resultado precisamente evidente. El hecho de poner en práctica el proyecto en un entorno real ha sido una experiencia realmente satisfactoria. Las pruebas realizadas, nos han permitido verificar experimentalmente que resulta mucho más conveniente enviar paquetes pequeños que paquetes grandes, sobre todo a través de internet.

Finalmente, este proyecto, nos ha llevado a desarrollar protocolos y estructuras para la correcta la transmisión, lo que, extrapolando este concepto, no solo nos permite el envío de audio digital, sino de cualquier tipo de información. Este hecho, nos abre nuevos retos que afrontar, lo que nos presenta la posibilidad de una extensa capacidad de ampliación del trabajo.

12.4. Ampliaciones

Lo primero que debemos destacar es que, como se ha comentado en la sección "3.4 Restricciones de diseño", la aplicación no encripta la información, con lo cual, se envían los datos en plano. Esto nos lleva a comentar una mejora interesante de la aplicación, que consistiría en la implantación de algoritmos de encriptación.

Al igual que ocurre con la encriptación, sería muy conveniente llevar a cabo tareas de compresión ya que, además de que el oído humano tiene un límite en el rango de frecuencias que puede oír, optimizamos el envío datos, evitando transmitir información inaudible.

La arquitectura del proyecto, ofrece bastante juego en cuanto a realizar nuevos desarrollos relacionados con nuevas formas de transmisión de información. El protocolo aquí implementado, puede servir para transmitir otro tipo de información como puede ser de vídeo. Por tanto, una posible ampliación podría ser la realización de videollamadas o de datos en tiempo real.

La inclusión de llamadas en grupo ha sido una mejora que no ha dejado de tenerse en cuenta a la hora de desarrollar el proyecto. Es posible que con pocas modificaciones se pueda llevar a cabo esta ampliación.

La inclusión de una ventana de chat y envío de documentos, puede resultar una extensión útil a la hora de compartir información entre los extremos.

Otra ampliación sería la inclusión de un sistema de suscripción con identificación mediante usuario y contraseña. De esta manera, podríamos almacenar los datos de la aplicación en la nube, tanto las llamadas como los contactos.

Referencias

Estudios

Asignatura *Diseño y configuración de Redes de Área Local* (2017).

Asignatura *Sistemas y Servicios en Red* (2016).

Internet (Consulta 2018)

[1] Skype: Información sobre la aplicación y sus características.
https://es.wikipedia.org/wiki/Skype
https://www.skype.com/es/home/

[2] WhatsApp: Referencia a la descripción de la aplicación y sus limitaciones.
https://es.wikipedia.org/wiki/WhatsApp

[3] Limitaciones de la versión de escritorio de WhatsApp.
https://elandroidelibre.elespanol.com/2016/05/whatsapp-en-windows-curiosidades.html

[4] Telegram: Definición y características de la aplicación Telegram.
https://es.wikipedia.org/wiki/Telegram_Messenger

[5] MTProto: Información sobre el protocolo empleado en la aplicación Telegram.
https://es.wikipedia.org/wiki/MTProto

[6] La aplicación Hangouts.
https://es.wikipedia.org/wiki/Hangouts
https://hangouts.google.com/?hl=es-419
https://chrome.google.com/webstore/detail/google-hangouts/nckgahadagoaajjgafhacjanaoiihapd

[7] La aplicación Line.
https://es.wikipedia.org/wiki/Line_(aplicaci%C3%B3n)
https://chrome.google.com/webstore/detail/line/ophjlpahpchlmihnnnihgmmeilfjmjjc?utm_source=chrome-ntp-icon

[8] Artículo sobre el funcionamiento de videollamadas en Line.
https://www.softonic.com/articulos/como-funcionan-las-videollamadas-en-line

[9] Artículo no oficial sobre el protocolo utilizado en la aplicación Line.
https://github.com/cslinmiso/LINE-instant-messenger-protocol/blob/master/line-protocol.md

[10] Artículo sobre audio digital.
https://www.animemusicvideos.org/guides/avtech/audio1.html

[11] Captura de señales de audio y preparación para su compresión.
https://riunet.upv.es/handle/10251/68934 (Oliver Gil, José Salvador)

[12] Qué es CG-NAT y por qué se utiliza
https://blog.masmovil.es/que-es-tecnologia-cgnat-masmovil/

[13] Unidad máxima de transferencia - MTU
https://es.wikipedia.org/wiki/Unidad_m%C3%A1xima_de_transferencia

Bibliografía

[14] James F. Kurose, Keith W. Ross. Redes de Computadoras - Un enfoque descendente.

[15] Andrew J. Wellings (2004). Concurrent and Real-time Programming in Java.

[16] Doug Lea. Programación Concurrente en Java: Principios y patrones de diseño.