UNIVERSIDADE DA CORUÑA

FACULTAD DE INFORMÁTICA

Departamento de Computación

PROYECTO FIN DE CARRERA
DE
INGENIERÍA TÉCNICA EN
INFORMÁTICA DE GESTIÓN

Servicio de talk verbal para entornos Linux

Autor: Riguera López, José
Tutor: Yáñez Izquierdo, Antonio

A Coruña, Julio de 2003

D. Antonio Yáñez Izquierdo

CERTIFICA

Que la memoria titulada **"Servicio de *talk* verbal para entornos Linux"** fue realizada por José Riguera López con D.N.I. 33.351.739-Z bajo su dirección. Esta documentación constituye la documentación que, con mi autorización, presenta el mencionado alumno para optar al grado de Ingeniero técnico en Informático de Gestión.

A Coruña, Julio de 2003

Firmado: Antonio Yáñez Izquierdo.

Especificación

Título del proyecto:	**Servicio de *talk* verbal para entornos *Linux***
Clase:	Proyecto clásico de ingeniería.
Nombre del alumno:	José Riguera López.
Nombre director:	Antonio Yáñez Izquierdo.
Nombre del tutor:	Antonio Yñez Izquierdo.
Miembros del tribunal:	
Miembros suplentes:	
Fecha de lectura:	
Calificación:	

A todos aquellos que se esfuerzan para que la información no conozca barreras;
a todos los que apuestan por el poder del conocimiento y de la palabra,
y a Vd, estimado lector, que ya con el sólo interés de leer estas líneas
provoca que el autor se sienta orgulloso de esta obra.

Agradecimientos

A Antonio Yáñez Izquierdo por aceptar y creer en este proyecto.

A Héctor Rivas por recomendarme no comer tantas pizzas,
a María Angueira por el nombre del programa,
a Rubén López por sus consejos.
y a todos aquellos que me conocen, por aguantarme estos últimos meses.

Resumen

El objetivo del presente proyecto es la creación de una aplicación de VoIP que permita a varios usuarios comunicarse entre sí usando ordenadores normales (PC's). El programa se ejecuta en plataformas Linux y, mediante una conexión bidireccional RTP/RTCP, permite mantener una conversación entre dos o más personas a modo de teléfono.

La aplicación tiene una arquitectura abierta y modular, que permite extender sus funcionalidades con tres tipos de plugins: E/S de audio, effectos sobre audio y codecs de compresión de voz. El programa es multisesión, es decir, puede mantener varias conversaciones activas simultáneamente, con o sin redes multicast. También está totalmente preparado para soportar la IPv6, la próxima generación de IP.

Ha sido desarrollado con Glib-2.0 lo que garantiza que el sistema es portable entre plataformas Unix e incluso -con pequeñas modificaciones- a plataformas MS Windows. La GUI se ha desarrollado en GTK+2.0 y es totalmente intuitiva, permite mostrar una imagen del usuario con el que está dialogando y dispone de una agenda en XML en la que gestionar los contactos: añadir, borrar, ignorar llamadas, etc.

Palabras clave

`VoIP, RTP/RTCP, IPv6, Glib/GTK+, multicast, multisesión, plugins, Linux`

Índice general

Capítulo 1

Introducción

Índice General

1.1. Prólogo

Tradicionalmente los servicios de telefonía y de datos han estado soportados por sistemas de redes distintos, basadas en tecnologías muy diferentes. Para el transporte de voz se usaban -y usan- principamente las redes telefónicas clásicas que emplean *señales analógicas* junto con técnicas de *conmutación de circuitos*. Por otra parte las redes de datos telemáticos son redes digitales y emplean una filosofía basada en la *conmutación de paquetes*, como ocurre, por ejemplo, con el protocolo de Internet (*IP*).

El desarrollo y maduración de las técnicas de transmisión de voz sobre redes de paquetes ha dado lugar a una fuerte tendencia hacia la integración del tráfico de voz en las redes de datos. En la actualidad es frecuente escuchar términos como "convergencia de redes" ó "convergencia de voz y datos" para denominar a esta tendencia. Las ventajas que ofrece la convergencia de redes han sido en un principio de índole económica. Tecnologías como *VoIP* hacen un uso mucho más eficiente del ancho de banda de la red, y permiten ahorrar en los costes de gestión y mantenimiento de redes, ya que solamente se utiliza una única red para ambos servicios.

Sin embargo, no sólo las razones económicas son las que justifican el interés en hacer converger las redes de voz y datos. La principal razón son las aplicaciones. La integración de redes facilita la creación de nuevas aplicaciones que comprenden voz y datos, como la mensajería unificada, que permitirá englobar bajo un único interfaz de usuario, accesible desde cualquier parte de la red, a todos los servicios a través de los cuales recibir mensajes (correo electrónico, fax, teléfonos, contestadores, etc.). O, por citar otros ejemplos, la integración de los centros de llamadas en los servidores web corporativos, que permitirán una atención rápida y especializada a los clientes; las aplicaciones de videoconferencia, la teleenseñanza, etc. Por otro lado, la utilización del protocolo *IP* facilita la creación de aplicaciones, ya que permite conectar la mayoría de los dispositivos; basta con desarrollar la aplicación una única vez y luego usarla en muchos tipos de redes diferentes. Aplicaciones que, aunque no técnicamente imposibles, serían de muy difícil realización sobre redes separadas.

Actualmente, gracias al avance tecnológico, ya es posible implantar, fácil y prácticamente sin costos, un sistema de *VoIP* en una red corporativa. Además, las empresas de telecomunicaciones (Cisco, Ericsson, etc) ofrecen pasarelas capaces de enlazar redes datos *IP* con las redes de telefonía tradicional basadas en centrales telefónicas de conmutación (*PBX*), permitiendo una implantación completa e integral.

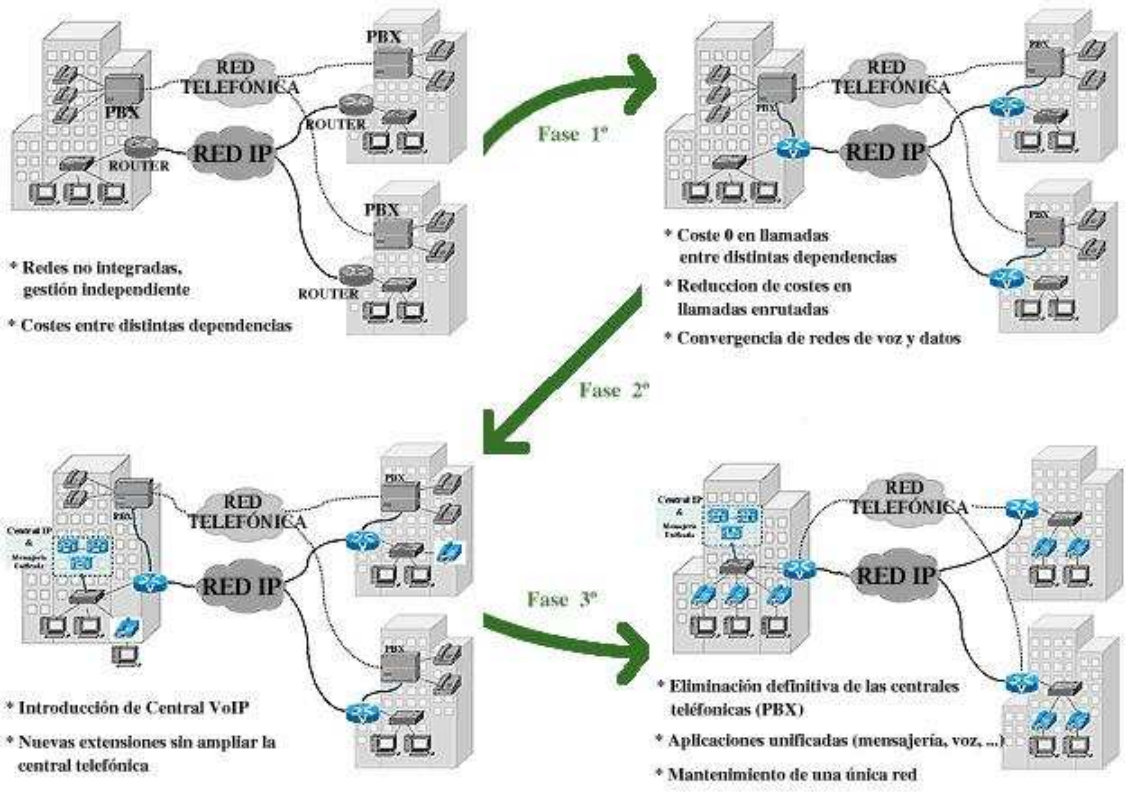

Figura 1.1: Fases del proceso de integración de las redes de voz y datos, con sus características y ventajas más destacables

1.2. Organización y contenido de esta memoria

Esta memoria se ha estructurado del siguiente modo:

Tras una breve introducción en la que se detalla la importancia de la integración de las redes de voz y datos, se procede a realizar un análisis de viabilidad (Capítulo 2), en donde se describe el proyecto -objetivos, motivaciones, recursos, riesgos, etc.- y las principales decisiones preliminares adoptadas. En ese mismo capítulo se analizan los antecedentes y las características, diseño, etc. de otras tecnologías u otras herramientas similares que van a rivalizar con este proyecto, es decir, se muestra el estado del arte.

A continuación se introduce al lector en las tecnologías elegidas para el desarrollo del proyecto (capítulo 3). Cabe resaltar que, debido a la multitud de tecnologías tratadas, no se profundizará demasiado en cada una, pero, se tratará lo mínimo para que el lector comprenda los conceptos empleados en el proyecto, el trabajo de ampliar los conocimientos se deja al interesado.

En el siguiente capítulo se muestra las especificaciones y requerimientos. En base a esas especificaciones se realiza un diseño (capítulo 5), se detalla la organización y esctructura del sistema, en donde se muestran los subsistemas clave del proyecto y sus inter-relaciones.

Una vez inciado el desarrollo técnico, se exponen los detalles de la codificación (capítulo 6). Se muestran todas las decisiones de implementación tomadas. Se explican qué algoritmos se aplicaron para resolver ciertos aspectos del diseño, etc. En el siguiente capítulo se muestran algunas de las pruebas realizadas durante el desarrollo del proyecto junto con detalles del rendimiento del programa.

Tras la implementación, se detallan los objetivos logrados, las ampliaciones realizadas al proyecto original, las posibles mejoras para versiones futuras, etc. También se exponen las conclusiones alcanzadas una vez finalizado este trabajo. Todo esto en el último capítulo 8.

Capítulo 2

Plan de proyecto

Índice General

Antes de la realización de un proyecto es necesario realizar una serie de análisis, más o menos profundos, dependiendo del tipo de sistema a desarrollar, para determinar su viabilidad y para intentar predecir los problemas que se encontrarán. Por ejemplo, si los problemas de desarrollo superan a las espectativas, no compensaría desarrollarlo. En este capítulo, se estudia el plan de proyecto para la propuesta mínima realizada, es decir, sin tener en cuenta las ampliaciones.

2.1. Análisis de viabilidad

Al emprender el desarrollo de un proyecto los recursos y el tiempo no son realistas para su materialización, sin tener pérdidas económicas ni frustración profesional. La viabilidad y el análisis de riesgos están relacionados de muchas maneras, si el riesgo del proyecto es alto, la viabilidad de producir software de calidad se reduce. En el presente capítulo se tratarán de definir estas áreas de interés:

Descripcion propuesta del proyecto.

Objeto : motivos, necesidades y ámbito de la realización del proyecto.

Objetivos : declaración del objetivo final del proyecto.

Requerimientos y recursos necesarios para el objetivo final.

Sistema de Control , seguimiento de los hitos obligados del proyecto.

Elementos de riesgo que pueden hacer inviable el proyecto.

Beneficios esperados tras la realización del proyecto.

Conclusiones : posibilidades del proyecto.

2.1.1. Descripción

El proyecto consiste en el desarrollo de una aplicación que use tecnología de *VoIP* para proveer un servicio de talk, parecido al que existe en entornos *Unix*, pero con la particularidad de ser un servicio verbal, es decir, la aplicación deberá permitir mantener una conversación entre dos personas. Para comenzar la conversación, el usuario indicará la dirección del otro interlocutor, y si éste existe y acepta, se establecerá una conexión entre ambos usuarios que les permitirá mantener un diálogo "full-duplex" (bidireccional), de la misma forma que ocurre en la línea telefónica tradicional. La dirección que especificará cada usuario tendrá el formato de *URL*, es decir, ***usuario@host[:puerto]***; por

ejemplo ***riguera@incertidumbre.dyndns.org[:10000]***.

Ambos usuarios podrán variar el volumen de los altavoces y la sensibilidad de sus micrófonos. Una consideración importante será el ancho de banda consumido por el proceso de comunicación y el protocolo elegido para realizar la conexión. El ancho de banda y el protocolo elegido son dos características que dependerán del algoritmo de compresión de audio seleccionado.

El software será desarrollado lo más modular posible, es decir, que use "plugins" que implementen diferentes algoritmos de compresión para la transmisión del audio, es decir, un desarrollo con una arquitectura abierta.

Toda la documentación y software del proyecto, una vez rematado éste, quedarán liberados bajo las licencias *GFDL*[1] y *GPL*[2], respectivamente.

2.1.2. Objeto

Este proyecto surgió debido a la necesidad establecer una comunicación fácil y directa entre un grupo de desarrolladores de software. Un programador, en ocasiones, necesita hablar con el resto del grupo o con el jefe del proyecto para aclarar conceptos y, a veces, por problemas de espacio, está en un local o despacho alejado. Aunque normalmente existe la posiblidad de usar el teléfono, el mantenimiento y/o la instalación de una central telefónica -o del propio telefóno- puede resultar demasiado costoso. También existen aplicaciones de mensajería instantánea, pero es difícil escribir y discutir una idea por este sistema. La aplicación anteriormente propuesta, podría ser capaz de solucionar gran parte de estas incomodidades. A continuación se detallan sus ventajas e inconvenientes.

Ventajas:

- No es necesaria la disponibilidad de una red de voz ni de terminales telefónicos.
- Mejor rendimiento de las infraestructuras. Permite un aprovechamiento de óptimo de las redes de datos ya instaladas en la empresa.
- Poca pérdida de tiempo, no es necesario conocer ni marcar ningún número teléfonico; la conexión es inmediata.
- Actividad poco absorbente, el usuario puede hablar y hacer otra cosa al mismo tiempo, como por ejemplo escribir.

[1] GNU Free Document License
[2] GNU General Public License

Inconvenientes, las desventajas son muy pocas y fácilmente salvables:

- Es necesaria una red IP.
- El PC debe tener instalada y configurada una tarjeta de sonido, junto con altavoces y micrófono.

2.1.3. Objetivos

En resumen, los **objetivos** mínimos propuestos para la finalización de todo el proyecto son, ordenados por prioridad:

1. Construir un programa capaz de comprimir y enviar el audio capturado por el micrófono de un PC a otro y simultáneamente reproducir el audio recibido del PC remoto. El ordenador remoto efectuará la misma operación.
2. El software deberá tener una interfaz gráfica de usuario sencilla e intuitiva que permita realizar fácilmente las conexiones.
3. Los usuarios podrán ajustar parámetros de audio tales como volumen y ganancia.
4. Opcionalmente se considerará la implementación de un subsistema que use "plugins" y/u otras mejoras en el software.
5. Realización de esta memoria.

Las principales **restricciones** que tendrá el proyecto son:

- El sistema se desarrollará y documentará empleando la metodología de análisis y diseño estructurado.
- El programa se ejecutará en máquinas de capacidad media (PII o superior) y con un sistema operativo con soporte multitarea *GNU/LINUX*, y en un entorno gráfico *Xwindows* (GNOME, KDE, etc)
- Tiempo limitado.

2.1.4. Requerimientos y recursos

Requerimientos, para alcancar el objetivo final es necesario:

- Un mínimo de dos PC's con el sistema operativo *Linux*. Se usan para recopilar información, analizar, diseñar, programar y probar el software. También es necesario que los PC's tengan sendas tarjetas de sonido configuradas y que estén conectados por red *IP*.

- Dos pares de altavoces o auriculares y un par de micrófonos. Son necesarios para captar y reproducir el audio.
- Acceso a documentación técnica sobre *Linux*, muestreo de audio, compresión de audio, *RFC's*, etc.

El tiempo máximo de elaboración asciende al período de un curso académico, aproximadamente 9 meses, con dedicación exlcusiva los últimos 6 meses y una única persona, que deberá planificar y realizar todas las fases del proyecto.

Recursos: Para la implementación del programa será necesario evaluar el empleo de diferentes tecnologías de voz sobre *IP* (*VoIP*) tales como *H323*, *SIP*, etc. Los principales recursos provienen de Internet: *papers*, artículos, foros, listas de correo electrónico, manuales de programación, ... También resultan imprescindibles las bibliotecas de la *UDC*[3], en especial, la biblioteca de la Facultad de Informática.

En cuanto a los recursos software, cabe destacar los siguientes:

- Distribución *Debian GNU/Linux* en su versión "testing" (Junio de 2003), conocida como "*Sarge*".
- Herramintas genéricas de programación, normalemente incluídas en las distribuciones de Linux. Algunas de las más usuales son:

  Compiladores GNU : ***gcc***, ***gpc***, ***g++***, etc. En función del lenguaje de programación elegido.

  Herramientas de depurado : ***gdb***, es el depurador clásico de entornos *Unix*; útil para corregir todos los fallos de accesos no permitidos a memoria y excepciones de punto flotante. ***Electric-fence*** es una biblioteca que substituye la función *malloc* de la librería estándar, encargada de realizar las reservas de memoria, por una versión propia que incluye bandas de memoria no válidas entre cada par de bloques de memoria reservada, lo que permite detectar más fácilmente desbordamientos de *buffer*. ***Memprof***, herramienta capaz de detectar fugas de memoria, es decir, bloques de memoria reservados dinámicamente y ya no puede ser accedido, lo que hace imposible su liberación.

  Herramientas de control de tráfico de red : ***tcpdump*** y ***ethereal***, permiten capturar parte del tráfico que pasa por una interfaz de red aplicando filtros que eliminen los paquetes no deseados para realizar un análisis posterior. Permiten corregir fallos en la implementación de los protocolos.

[3] Universidade da Coruña

- Páginas del manual, documentan las funciones, librerías y llamadas del sistema.

En general todos los recursos software empleados se caracterizarán por ser "open source", es decir estar bajo la licencia *GPL*. Normalmente todos estos recursos ya vienen integrados en las distribuciones de *Linux*; posteriormente se indicará más detalladamente las herramientas empleadas.

2.1.5. Sistema de Control

El sistema de control de este proyecto está basado en un calendario de actuación, es decir, simplemente se controla el tiempo invertido en cada una de las fases o hitos. El tiempo máximo de realización es de 9 meses, con fecha limite de finalización el 4 de Julio de 2003. La fecha de la propuesta de este proyecto fue a finales de Mayo de 2002, si bien, no fue iniciado hasta Septiembre de 2002. Los hitos que comprende son:

1. **Documentación y viabilidad**: consiste en acaparar información de proyectos similiares, se buscan posibles tecnologías para el desarrollo del proyecto, etc. La fase acaba con un análisis de viabilidad y, si éste es positivo, se elabora un *Plan de proyecto*. En el caso de este proyecto, al ser de *tipo B* -es decir, propuesto por el alumno- esta fase se elaboró antes de la propuesta, ya que el alumno no va a proponer un proyecto inviable, por esa razón el tiempo invertido en esta fase no entra dentro del tiempo máximo de realización, pero, se estima en 4-5 semanas.

2. **Análisis**: tras la *Documentación*, se procede al análisis de toda la información, se evalúa el uso de distintas tecnologías, se eligen las más adecuadas y se trazan las líneas maestras del proyecto. Tiempo estimado: 4-5 semanas.

3. **Diseño, implementación y pruebas**: durante el *Diseño* se establece la arquitectura del software, es decir, la definición de los subsistemas y subobjetivos. En conjunto, todos los subsitemas deben cumplir los requerimientos. El tiempo máximo de realización es de 1 mes. La *Implementación* del software, se basa en el *Diseño*: se implementa cada subsistema, se comprueba que cumple con los requisitos y se elaboran pruebas específicas. Esta fase está estrechamente relacionada con la anterior, de forma que el diseño original puede sufrir modificaciones.

4. **Memoria**, es el documento final que analiza todas las fases del desarrollo del proyecto. Tiempo disponible para esta fase, 4 semanas.

Este proyecto sigue un ciclo lineal, excepto en las fases de diseño, implementación y pruebas, el que el ciclo es evolutivo. Dicho de otro modo, las fases lineales son *Documentación* y *Análisis*, esto significa que ninguna de estas fases puede comenzar hasta que no

haya terminado la anterior. Al final de cada una de estas dos fases se elabora un plan que define la actuación en fases sucesivas. El ciclo evolutivo engloba el *Diseño, implementación y pruebas*, ya que la etapa de diseño puede sufrir modificaciones en función de etapas sucesivas, y también, debido a las posibles ampliaciones del proyecto.

2.1.6. Elementos de riesgo

El principal riesgo del proyecto es el no cumplimiento de los hitos anteriores en el tiempo indicado y considerando que los plazos de entrega son fijos. Otros posibles riesgos proceden de la utilización de librerías externas, que pueden presentar fallos de programación ("bugs") . Por estas circunstancias, los plazos de las distintas fases por las que atraviesa el proyecto, pueden acumular un retraso máximo total de 1 mes.

2.1.7. Beneficios

El objetivo de este proyecto no es conseguir beneficios económicos, simplemente se busca finalizar un ingeniería técnica, por otro lado, este proyecto quedará liberado bajo la *GPL*, por lo que no tiene sentido plantearse beneficios económicos.

Sin embargo, las espectativas del este proyecto son altas. Existen cientos de previsiones que auguran crecimientos exponenciales al número de minutos hablados a través de VoIP, una explosión similar a la del uso del correo electrónico a mediados de la década de los noventa. La siguiente gráfica (figura 2.1) resume un estudio de Philips realizado en el año 2000, en la que se representa el porcentaje del total de tiempo de conversación que se realizará usando VoIP.

Sin embargo, queda todavía por clarificar en este mercado el papel de las grandes operadoras de telefonía "tradicional": la transmisión de voz sobre redes *IP* supone una seria amenaza para su modelo de negocio, y no hay que olvidar que generalmente son ellas las que proporcionan la conectividad a Internet. Parece por lo tanto poco factible que no usen su posición de fuerza para impedir el uso de estos servicios.

2.1.8. Conclusiones

La viabilidad y el análisis de riesgo están relacionados de muchas maneras. Si el riesgo del proyecto es alto la viabilidad de producir software de calidad se reduce. Las conclusiones extraídas se agrupan en tres partes diferenciadas:

Viabilidad económica. En el caso de este proyecto no es necesario un estudio de viabilidad económica ya que los beneficios ni la justificación económica no existe. Se trata de un proyecto hecho con software libre y para ser "open source".

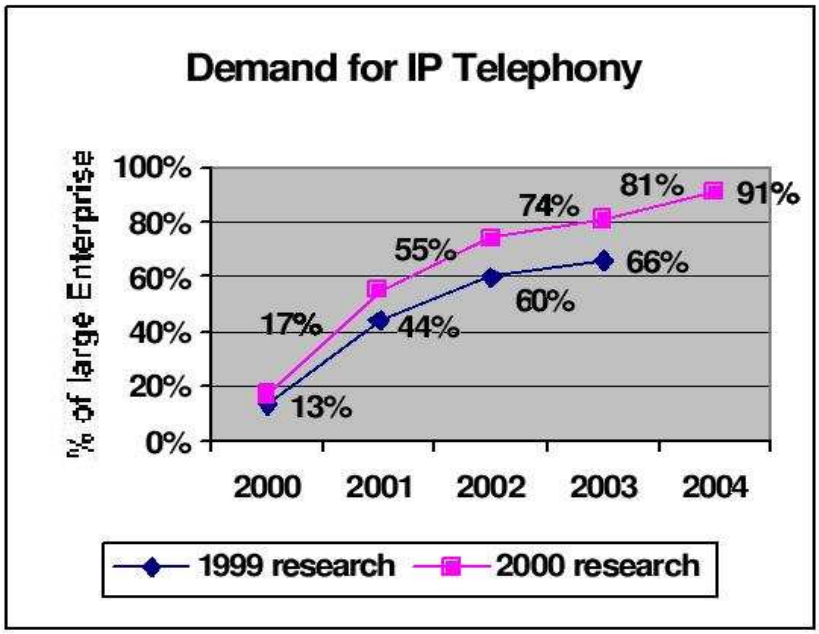

Figura 2.1: Espectativas de para el crecimiento de VoIP.

Viabilidad técnica. La viabilidad tecnológica es frecuentemente el área más difícil de valorar en esta etapa del proceso. Es esencial que el proceso de análisis y definición se realice en paralelo con una valoración de la viabilidad técnica. De todas formas, en este caso la viabilidad técnica es positiva, puesto que el riesgo técnico es bajo, como lo demuestra la existencia de proyectos similares, que más adelante serán comentados en la seccion de *Antecedentes*.

Viabilidad legal. No se esperan problemas legales, aunque la existencia de patentes de software en algunos países, junto con las restricciones a la implementación de determinados algoritmos de cifrado en otros, puede limitar el uso del software. Los problemas relacionados con sistemas cifrados pueden evitarse haciendo varias versiones del software y/o con compilación condicional.

2.2. Antecedentes

En esta sección se explicarán las principales soluciones que se han dado al problema de la telefonía sobre *IP* en general. Luego se comparará el proyecto con otros de propósito similar que existen en este momento.

2.2.1. Estado del arte

Debido a la filosofía de Linux, y en general de todo el software libre, tardaron en aparecer aplicaciones que usaran técnicas *VoIP* para transmitir audio. La filosofía "open source" choca frontalmente con las patentes de software y los estándares cerrados. Hasta antes del 2000, sólo existía un proyecto de *VoIP* que funcionaba en Linux con unas características aceptables, se trataba de **Speak Freely**. Esta situación estaba provocada porque sólo había un estándar disponible, el *H.323* de la *ITU*, y para acceder a la descripción del *H.323* es necesario abonar una cantidad de dinero a la *ITU-T*. Otros inconvenientes de *H.323* son: complejidad -está formado por otros protocolos-; es prácticamente cerrado y algunos códecs de compresión de voz que usa son propietarios. A finales del 2000, el grupo *MMUSIC* del *IETF*, estandarizó un nuevo protocolo, el *SIP* (Session Initiation Protocol). Este protocolo, a diferencia del anterior es mucho más ligero; su estándar es abierto -está disponible en Internet- y todavía se encuentra en estado de desarrollo y revisión. *SIP* Es un protocolo muy simple y fácil de implementar y a diferencia de *H.323*, que define todo el proceso de comunicación de audio, *SIP*, sólo define como se debe iniciar la sesión; la transmisión de audio es independiente de *SIP*.

A partir de la publicación de *SIP* (primavera 2001) empezaron a surgir muchos proyectos de *VoIp* para Linux, FreeBsd, etc. que apostaron decididamente por *SIP* frente a *H.323*, esa es la razón por la que (casi) todos los proyectos "open source" emplean SIP junto con RTP.

RTP es el otro protocolo en liza. Es un protocolo que se emplea para transportar el audio comprimido a través de una red (*IP*), pero, para conocer las característas -formato de compresión, puertos de recepción y envío, etc.- de esa transmisión es nesario *SIP* ó *H.323*. Por esta razón, antes del año 2000, era muy complicado usar **Speak Freely**, ya que no usaba ninguno de estos dos protocolos. Para iniciar una conversación con **Speak Freely**, ambos usuarios debían conocer y configurar los parámetros de RTP y del audio comprimido.

En el capítulo 3 se detallará una introducción *H.323*, *SIP* y *RTP* y se analizarán todas sus características.

2.2.2. Proyectos similares

Los más importantes se detallan a continuación. Las características de cada uno de ellos están analizadas a fecha de Junio de 2003.

Linphone, creación de Simon Morlat, tiene licencia *GPL* y permite realizar de manera fácil y eficiente llamadas de voz sobre *IP* desde el entorno de *GNOME* a través de Internet o redes locales entre estaciones GNU/Linux y hacia otros soft-phones compatibles con SIP. Estas son las características de **linphone**:

- Desarrollado para linux -aunque no se descarta el funcionamiento en otras plataformas *Unix*- y usando el entorno *Gnome*, pero puede ejecutarse en *KDE*, además, a partir de la version 0.9.0, está disponible una versión para consola llamada ***linphonec***.
- Interfaz gráfica fácil de usar, como un teléfono celular, con sólo dos botones, ya que incorpora una agenda en la que guardar los contactos.
- Incluye una gran variedad de codecs compresores de audio: ADPCM. GSM, Speex, etc. Estos codecs están integrados en el código del programa.
- Utiliza el protocolo *SIP*. *SIP* es un Protocolo de Inicio de Sesion (Session Initiation Protocol) que está estandarizado por la *IETF*[4].
- Es software libre, liberado bajo los términos de la *GPL*.

El software está integramente programado en C, con técnicas de OO (Orientación a Objetos). Tiene una librería independiente llamada *oRTP* (Open RTP) que implementa el protocolo RTP, desarrollada usando POO y paralelamente al programa **linphone**. Sin embargo todavía no soporta cifrado de RTP ni el protocolo RTCP. La página web del proyecto es *http:www.linphone.org*.

Speak Freely. Tal vez sea uno de los primeros programas creados para este menester -iniciado en 1991 por John Walker, fundador de *Autodesk*- es, por ello, uno de los más populares. Existen versiones para sistemas *Unix* -tanto *System V* como *BSD*- y Windows. Las características más destacables de **Speak Freely** son:

- Multiplataforma, existen versiones tanto para sistemas *Unix* como *Windows*.
- Interoperabilidad con ICQ, un servicio de mensajería instantánea.
- Modo de conferencia, que permite charlar con varias personas a la vez, sin necesidad de estar en redes *multicasting*.
- Permite grabar mensajes de voz que informen a la persona llamante de que no está disponible.

[4]Internet Engineering Task Force, define los estándares de Internet (http://www.ietf.org)

- Soporta cifrado de las sesiones. Implementa varios algoritmos de cifrado: DES, IDEA, PGP, Blowfish, etc.
- Aparte de *RTP*, implementa el protocolo *VAT* (Visual Audio Tool), un protocolo muy simple de streaming de audio en tiempo real creado en *Lawrence Berkeley Laboratory* (http//:www.lbl.org). Este protocolo no está estandarizado por ninguna organización.
- Soporta estos codecs compresores de audio: DVI4, GSM, L16, LPC, PCMA y PCMU. Están integrados dentro del programa.
- Es software libre, liberado bajo los términos de la *GPL*.

Las principales características de este programa son ser multiplataforma -la versión 6.0 puede ser ejecutada en plataformas de 16 bits como *OS/2 Warp* y tiene total interoperabilidad con las restantes, anteriores y posteriores- y la posibilidad de tener varias sesiones con distintos usuarios (modo de conferencia). La versión actual (7.0) está programada en *C*, si bien, los desarrolladores planear reescribir el código en *C++* para la version 8.0 y futuras. También existen versiones sin algoritmos criptográficos, por restricciones legales en algunos países. Más información en *http://www.speakfreely.org*.

RAT - Robust Audio Tool. Como su propio nombre indica es, probablemente el programa más robusto para conferencia de audio. Las características más importantes son:

- Multiplataforma, FreeBSD, HP-UX, IRIX, Linux, NetBSD, Solaris, SunOS y Windows 95/NT.
- Incorpora técnicas de control de detección de voz, detección de ruido, etc.
- Multiconferencia, que permite charlar con un grupo de personas, pero necesita redes *multicasting*.
- Completamente compatible con el estándar RTP/RTCP, puede incluso enviar audio redundante para contrarestar pérdidas de paquetes.
- Soporta cifrado de sesiones RTP/RTCP con DES.
- Soporta estos codecs compresores de audio: GSM, ADPCM (G.726), G.711 y LPC. Están integrados dentro del programa.
- Es software libre, liberado bajo los términos de la *GPL*.

Este programa se diferencia de los anteriores en que soporta el protocolo RTP/RTCP íntegramente. También tiene varios sistemas de reparación de audio, lo que permite que aunque se pierdan paquetes de audio el usuario no note esas pérdidas. Las estrategias de reparación de audio son:

Envio redundante de audio. Cada vez que se envia un paquete, se aprobecha para enviar los n anteriores, conjuntamente. De esta forma, si se pierde algún paquete, sólo es necesario esperar al siguiente, el cual contendrá el audio para el instante actual y para n instantes anteriores. El problema se produce cuando se pierden n paquetes o más, no existen posibilidades de reparación. El inconveniente de este sistema es que multiplica por n el ancho de banda requerido y, en el peor de los casos, se puede producir un retardo de varios segundos en el audio recibido.

Entrelazado de audio Una vez formado un paquete, se divide en n trozos, forma un nuevo paquete compuesto por n trozos ordenados secuencialmente de los paquetes originales, y se envia el paquete compuesto. Para recomponer un paquete, es necesario esperar la llegada de n. La ventaja de este sistema es que no aumenta el ancho de banda consumido, pero la conversación sufre un retardo constante de varios segundos, debido a la necesidad de esperar a n paquetes. Si se pierde un paquete el usuario no notará nada, ya que afectará mínimamente a los n paquetes sucesivos.

Esta aplicación también puede hacer streaming de audio. Más información en *http://www-mice.cs.ucl.ac.uk/multimedia/software/rat*.

Capítulo 3

Análisis tecnológico

Índice General

Este capítulo sirve de pequeña introducción técnica de las distintas tecnologías empleadas en el desarrollo de este proyecto.

3.1. Tratamiento del sonido

Esta sección pretende introducir al lector una serie de conocimientos básicos acerca de la naturaleza del sonido. En un principio se presentan una serie de conceptos físicos claves para enterder su naturaleza, propagación, percepción por el oído humano. Más adelante se aborda el procesado digital del sonido y su tratamiento desde el punto de vista del procesamiento de señales, esto es, como el sonido es captado por un dispositivo digital como un muestreador DSP para ser tratado y procesado. La terminolgía empleada en esta sección es clave para enterder capítulos posteriores.

3.1.1. Acústica

Se entiende por acústica la parte de la física que trata del sonido y de todo lo que a él se refiere.

Sonido

Desde el punto de vista físico el sonido es una vibración mecánica en un gas, líquido o medio sólido. La propiedad elástica del medio permite al sonido propagarse en forma de onda desde la fuente, algo parecido a las ondas que se generan en un charco al arrojar una piedra. Cada vez que un objeto vibra, una pequeña porción de energía se pierde en forma de sonido, o expresado de otro modo, la vibración provoca un transvase de energía al medio -normalmente el aire- que da lugar al sonido.

El sonido se propaga en el aire en forma de variación de presión. Un altavoz, por ejemplo, transmite las variaciones de presión al aire y sólo esta variación se mueve, no el aire; en la comparación del charco mencionada antes, las ondas se desplazan por el agua, mientras que ésta permanece en el mismo lugar, pero oscilando verticalmente. En el aire las ondas de sonido se propagan a 340 metros por segundo en condiciones normales. En la figura 3.1 se muestra la propagación de la onda en un medio que podría ser el aire.

El parámetro Z es la amplitud de onda, es le valor máximo que puede alcanzar. El parámetro T_0 es el período, se expresa en segundos; T_0 es igual al inverso de la frecuencia F. La frecuencia se expresa en *Hercios* (Hz).

A continuación se va a definir un modelo matemático que describe la naturaleza del sonido. Cualquier sonido genérico se puede formar a partir de una suma de movimientos

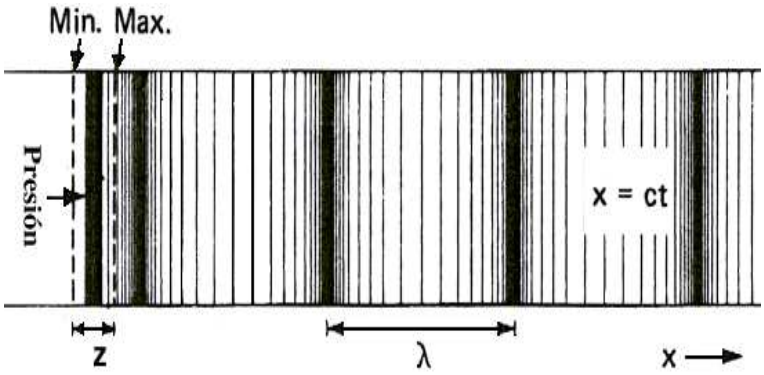

Figura 3.1: Transmisión de la vibración en un fluido

armónicos simples, por ello se van a definir las características del movimiento armónico. Si se representa la figura 3.1 como una onda se obtiene la figura 3.2

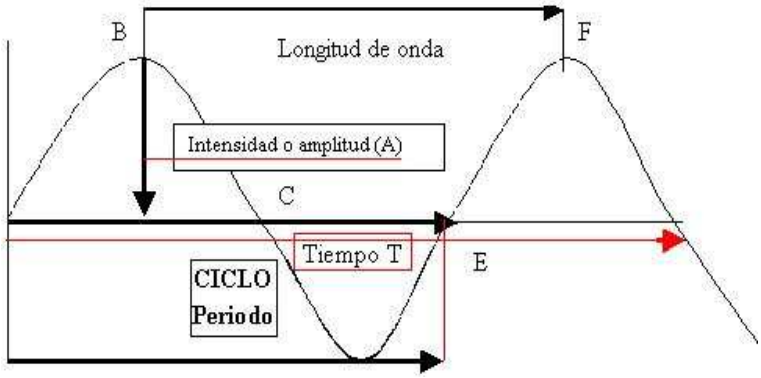

Figura 3.2: Movimiento armónico simple

Como toda onda se puede considerar como superposición de ondas sinusoidales de frecuencia, amplitud y fase correspondientes (*Teorema de Fourier*), un sonido complejo se puede descomponer como una suma de armónicos simples de distinta frecuencia.

Las ondas sonoras capaces de ser detectadas por el oído humano van desde 20 Hz (umbral inferior) a 20000 Hz (umbral superior). Por debajo de 20 Hz están los infrasonidos (mareas, ondas sísmicas) y por encima de 20000 Hz, los ultrasonidos (como el sonar, de baja energía, y las vibraciones de las redes cristalinas (cuarzo), de alta energía.

El ruido es un sonido audible no armonioso. Procede de ondas no periódicas. Una nota musical es un sonido agradable; procede de ondas periódicas. El sonido o nota fundamental es la víbración cuya frecuencia F_0, es la más baja que se puede obtener en la flauta

aguda (caramillo). Un armónico es una nota cuya frecuencia es un múltiplo entero de F_0.

Las notas o sonidos musicales se caracterizan por la intensidad, el tono y el timbre. La intensidad de un sonido depende de la mayor o menor amplitud de la onda, ya que la energía de la misma es función del cuadrado de la amplitud. La audición está unida a la intensidad de la onda sonora. Para cada frecuencia hay una intensidad mínima (umbral de audición) por debajo de la cual no se oye; y una intensidad máxima que produce sensación de dolor (umbral doloroso). El nivel de intensidad sonoro, se mide en decibelios (dB) que se explicaran más adelante. El tono de un sonido depende de su frecuencia. Los tonos agudos tienen mayor frecuencia que los graves. De la relación: $f = v/l$ se deduce que a mayor longitud de onda, menor frecuencia, y viceversa. El timbre depende de los armónicos que acompañan a los sonidos; como éstos varían con los instrumentos, por el timbre se distingue una nota dada por diferentes instrumentos.

Amplitud

La medida de la amplitud de una onda es importante porque informa de la fuerza, o cantidad de energía, de una onda, que se traduce en la intensidad de lo que oímos, su unidad de medida es el decibelio. Un decibelio, abreviado como dB, es una unidad de medida de la fuerza de la señal y es útil en la comparación de la intensidad de dos sonidos. La sensibilidad del oído humano es extraordinaria, con un rango dinámico o variación en intensidad muy amplio. La mayoría de los oídos humanos pueden capturar el sonido del murmullo de una hoja y, después de haberse sometido a ruidos explosivos como los de un avión, siguen funcionando. Lo que es sorprendente es que la fuerza de la explosión en un avión es al menos 10 millones de veces mayor que el murmullo que una hoja produce con el viento.

Según la *ley de Weber-Fechner*, el efecto sobre el oído de un cambio de intensidad depende de la intensidad que precede al cambio, por ello el oído necesita un porcentaje elevado de variaciones en la fuerza de un sonido para detectar un cambio en la intensidad percibida. Una forma simple de comprender esto es comenzar con una nota de determinada intensidad, por ejemplo 10 unidades, incrementarla luego a 100 y después a 1000 unidades. Estos dos cambios los interpretaría el oído como idénticos en potencia, puesto que la proporción de 100/10 es igual a 1000/100. Otro modo de expresar esto es escribir los valores de las intensidades en potencias de diez: 10^1, 10^2, 10^3 y se aprecia que los cambios iguales en potencia vienen dados por cambios iguales al logaritmo de la intensidad. El oído tiene una respuesta logarítmica y en mediciones de acústica se emplea una unidad

llamada *belio*, abreviado B . Si la intensidad inicial I_0 se incrementa hasta un nuevo valor I_1 se tiene:

$$belios = log_{10}\left(\frac{I_1}{I_0}\right)$$

En la práctica, el belio es demasiado grande y por ello se emplea el *decibelio* (dB). La relación en decibelios será entonces:

$$dB = 10 \cdot log_{10}\left(\frac{I_1}{I_0}\right) \tag{3.1}$$

donde I_0, es la intensidad inferior de audición que se toma como punto de referencia y en el aire vale: $I = 10^{-12}W_m^{-2}$. Así un sonido cuya intensidad sea 1000 veces superior a I_0 tiene de nivel de: $10log(10^3 10^{-22}/10^{-12}) = 30dB$.

Los logaritmos representan un recurso muy útil de cálculo, y a esto se debe que el decibelio se use universalmente en ingeniería electrónica para comparar dos potencias eléctricas. Así, por ejemplo, si la potencia de entrada de un amplificador es W_0 y la de salida W_1, la amplificación en decibelios será:

$$A = 10 \cdot log_{10}\left(\frac{W_1}{W_0}\right)$$

En acústica se suele tratar con mayor frecuencia con presiones que con intensidades, y con dos presiones, la relación en decibelios es:

$$dB = 20 \cdot log_{10}\left(\frac{P_1}{P_0}\right) \tag{3.2}$$

donde P_0 es la presión inicial y P_1 la presión con la que se quiere relacionar. Esta relación se obtiene a partir de la fórmula 3.1 y considerando que $intensidad = f(presion^2)$:

$$\begin{aligned} dB &= 10 \cdot log_{10}\left(\frac{P_1^2}{P_0^2}\right) \\ &= 20 \cdot log_{10}\left(\frac{P_1}{P_0}\right) \end{aligned} \tag{3.3}$$

En el aspecto práctico de la amplitud, un incremento de sólo 3 dB duplica la intensidad de un sonido. Por ejemplo, un sonido con 86 dB tiene, el doble de fuerza que un sonido con 83 dB y cuatro veces más que un sonido con 80 dB. Desde la perspectiva de nuestra

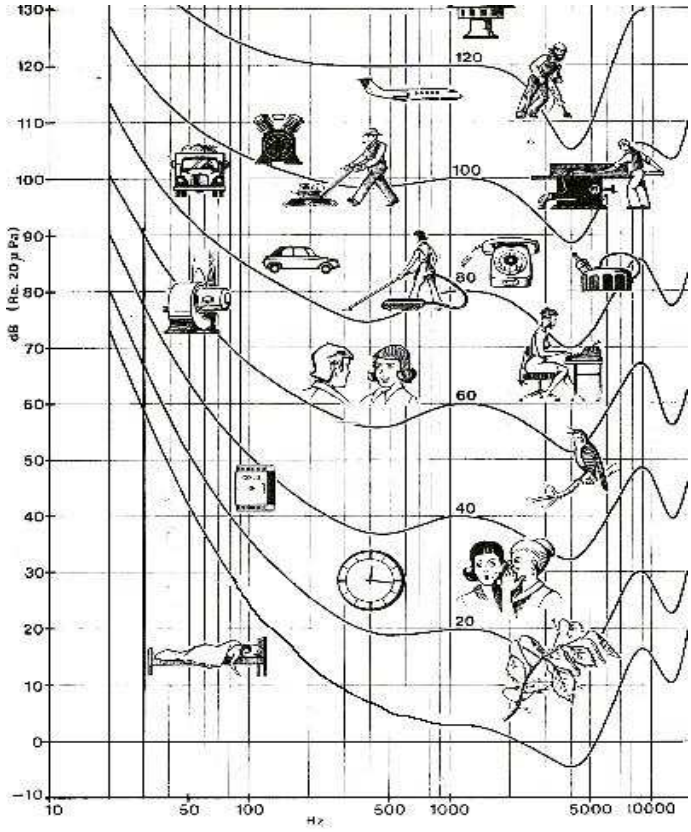

Figura 3.3: dB de varias fuentes de sonido

percepción de la intensidad, un incremento de 3 dB, que da lugar a que se duplique la fuerza, provoca que el sonido se perciba sólo ligeramente más alto. Es necesario un aumento en 10 dB para que nuestros oídos perciban un sonido con el doble de intensidad. La figura 3.3 muestra el nivel en dB de varias fuentes de sonido usuales.

Rango dinámico

El rango dinámico es el variación de frecuencias que puede alcanzar o recorre una señal de audio. Es una característica que influye notablemente en la calidad de los sonidos grabados. El rango dinámico de un sonido reproducido nunca es comparable al real. La razón principal es que un sistema -analógico o digital- no puede duplicar el rango dinámico completo de, por ejemplo, una orquesta o de un concierto de rock. Una orquesta puede alcanzar los 110 dB en su climax y en el punto más suave bajar hasta los 30 dB, dando lugar a un rango dinámico de 80 dB. Este rango es superior al rango dinámico de un sistema estéreo típico y, de hecho, superior a la capacidad de grabación de medios tales como un disco de vinilo y una cinta de audio. El programa, por ejemplo, tiene un rango

dinamico (interno) de 96 dB, ya que trabaja con audio en formato 16 bits y partiendo de la ecuación de cálculo de dB 3.2, se obtiene :

$$Rango\,dinamico\ = 20 \cdot log_{10}\left(R_{\Delta}\right) = 20log_{10}(2^{16}) = 96\,dB$$

En la siguiente lista se muestra rango dinámico varios sistemas de almacenamiento de audio:

- Disco de vinilo: 65 dB
- Cinta magnetofónica: 55 dB
- Cd-Audio (16 bits): 96 dB
- Muestreo 8 bits: 48 dB

Ancho de banda

Existe una medida estándar para definir el ancho de banda: el rango de frecuencias sobre el que la amplitud de la señal no difiere del promedio en más de 3 dB, es decir la diferencia de las frecuencias en la que se produce una caída de 3 dB, ya es el punto donde su amplitud cayó a la mitad, y éste es el mínimo cambio en la fuerza de la señal que puede ser percibido como un cambio real en la intensidad por la mayoría de los oídos.

A menudo el ancho de banda se simboliza mediante un único número cuando la frecuencia baja está bastante próxima a cero. Por ejemplo, el ancho de banda de una voz femenina se sitúa en torno a los 9 kHz, aunque realmente puede estar en el rango que va desde los 200 Hz hasta los 9 kHz.

Es importante tener en cuenta que el ancho de banda de un sistema de sonido depende del enlace más débil del canal, que normalmente no es la tarjeta de sonido. La calidad del sonido producido por el PC refleja el esfuerzo de muchos componentes, y la salida no será mejor que la interpretación del miembro menos capacitado de un grupo. En el caso del sistema de sonido del ordenador, una señal debe pasar por muchas fases de transformación de audio y por diferentes dispositivos. Por ejemplo, el sonido grabado mediante un micrófono y que luego es reproducido. La tarjeta de sonido transforma el sonido recogido del micrófono en una señal eléctrica que, posteriormente, se transforma en audio digital y se almacena en disco. El audio digital del disco es transformado de nuevo en una señal eléctrica y reproducido a través de los cascos o de los altavoces. El ancho de banda efectivo del sistema de sonido está limitado por el dispositivo con el ancho

de banda más estrecho de todos los dispositivos que procesan el sonido. El enlace más débil en grabación suele ser el micrófono, que tiene probablemente un ancho de banda aproximadamente de 12 kHz.

3.1.2. Psicoacústica

El oído humano percibe un rango de frecuencias entre 20 Hz. y 20 Khz. En primer lugar, la sensibilidad es mayor en la zona alrededor de los 2-4 Khz., de forma que el sonido resulta más difícilmente audible cuanto más cercano a los extremos de la escala. En segundo lugar está el enmascaramiento, cuyas propiedades utilizan exhaustivamente los algoritmos más interesantes: cuando la componente a cierta frecuencia de una señal tiene una energía elevada, el oído no puede percibir componentes de menor energía en frecuencias cercanas, tanto inferiores como superiores. A una cierta distancia de la frecuencia enmascaradora, el efecto se reduce tanto que resulta despreciable; el rango de frecuencias en las que se produce el fenómeno se denomina banda crítica. Las componentes que pertenecen a la misma banda crítica se influyen mutuamente y no afectan, ni se ven afectadas, por las que aparecen fuera de ella. La amplitud de la banda crítica depende de la frecuencia y viene dada por unos determinados datos que demuestran que es mayor con la frecuencia. Hay que señalar que estos datos se obtienen por experimentos psicoacústicos, que se realizan con expertos entrenados en percepción sonora, dando origen con sus impresiones a los modelos psicoacústicos.

Lo descrito es el llamado enmascaramiento simultáneo o en frecuencia. Existe, asimismo, el denominado enmascaramiento asimultáneo o en el tiempo, así como otros fenómenos de la audición que no resultan relevantes en este punto. La idea es que ciertas componentes en frecuencia de la señal admiten un mayor ruido del que generalmente consideraríamos tolerable y, por tanto, requieren menos bits para ser codificadas si se dota al codificador de los algoritmos adecuados para resolver máscaras.

Otra consideración importante en las comunicaciones de audio es la calidad de la transmisión. Si la calidad es muy buena y se usan técnicas de transmisión discontinúa o DT (Discontinous Transmission) apoyadas por VAD (Voice Activity Detection o detección de voz) provoca que los interlocutores tengan a veces la sensación de que se ha perdido la comunicación y aparecan expresiones del tipo “sigues ahí”, “me escuchas”, etc. Esto se soluciona generando lo que se conoce como *ruido de comfort* mediante el sistema CNG (Comfort Noise Generation). El ruido de comfort se genera en el propio terminal, es un engaño del sistema, y consiste en generar un ligero ruido gausiano de fondo, pero sin entorpecer la comunicación.

3.1.3. Modulación de audio

La tecnología digital es más avanzada y ofrece mayores posibilidades, menor sensibilidad al ruido en la transmisión y capacidad de incluir códigos de protección frente a errores, así como cifrado. Con los mecanismos de decodificación adecuados, además, se pueden tratar simultáneamente señales de diferentes tipos transmitidas por un mismo canal. La desventaja principal de la señal digital es que requiere un ancho de banda mucho mayor que el de la señal analógica, de ahí que se realice un estudio en lo referente a la compresión de datos.

El proceso de digitalización se compone de dos fases: muestreo y cuantización. En el muestreo se divide el eje del tiempo en segmentos discretos: la frecuencia de muestreo será la inversa del tiempo que medie entre una medida y la siguiente. En estos momentos se realiza la cuantización, que, en su forma más sencilla, consiste simplemente en medir el valor de la señal en amplitud y guardarlo. El teorema de Nyquist garantiza que la frecuencia necesaria para muestrear una señal que tiene sus componentes más altas a una frecuencia dada f es como mínimo $2f$. Por tanto, siendo el rango superior de la audición humana en torno a los 20 Khz, la frecuencia que garantiza un muestreo adecuado para cualquier sonido audible será de unos 40 Khz. Concretamente, para obtener sonido de alta calidad se utilizan frecuencias de 44'1 Khz (CD Audio). Otros valores típicos son submúltiplos de la primera, 22 y 11 Khz. Según la naturaleza de la aplicación, por supuesto, las frecuencias adecuadas pueden ser muy inferiores, de tal manera que el proceso de la voz acostumbra a realizarse a una frecuencia de entre 6 y 20 Khz. En lo referente a la cuantización, es evidente que cuantos más bits se utilicen para la división del eje de la amplitud, más "fina" será la partición y por tanto menor el error al atribuir una amplitud concreta al sonido en cada instante. Por ejemplo, 8 bits ofrecen 256 niveles de cuantización y 16, 65536. El margen dinámico de la audición humana es de unos 100 dB. La división del eje se puede realizar a intervalos iguales o según una determinada función de densidad, buscando más resolución en ciertos tramos si la señal que se trata tiene más componentes en cierta zona de intensidad.

La modulación por impulsos codificados (PCM) es el método más común de codificar una voz analógica en un flujo digital. El proceso de PCM es el siguiente:

1. Las formas de onda analógicas se pasan por un filtro de frecuencia de voz para filtrar cualquier cosa que sea mayor de 4000Hz. Siguiendo el teorema de Nyquist, se necesita muestrear a 8000 muestras por segundo para alcanzar una transmisión de voz de buena calidad.

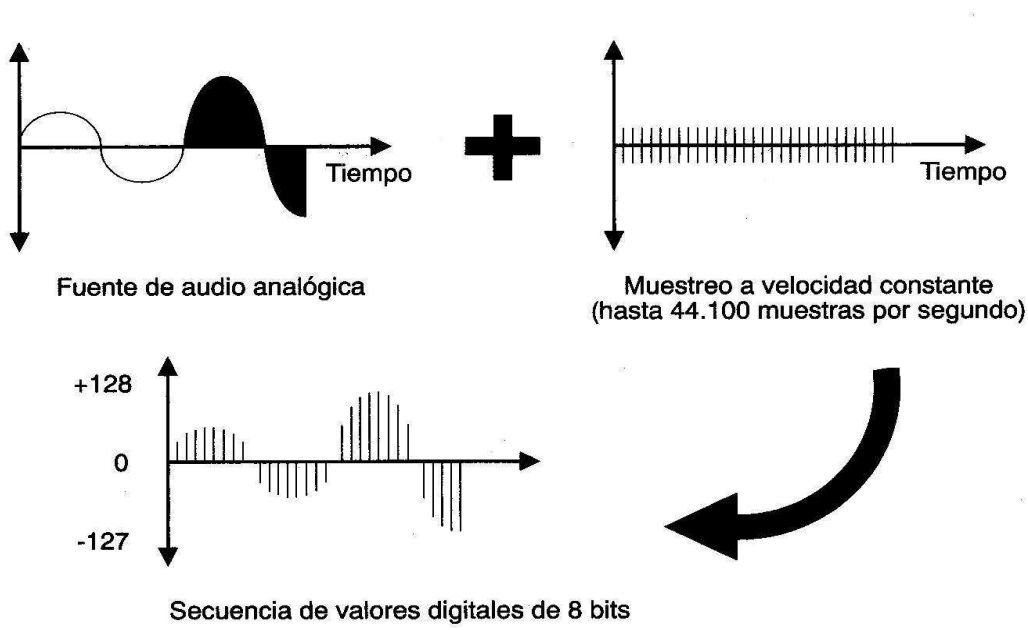

Figura 3.4: Conversión Analógico-Digital del sonido (ADC)

2. La señal analógica filtrada es luego muestreada a una velocidad de 8000 veces por segundo.

3. Cuando se ha muestreado la forma de onda, ésta se convierte en una onda digital discreta. Esta muestra está representada por un código que indica la amplitud de la forma de onda en el instante en que se tomó la muestra. La forma de telefonía de PCM utiliza 8 bits para el código y un método de compresión logarítmico que asigna más bits para señales de amplitud más baja.

   Si se multiplican las palabras de 8 bits a 8000Hz, se obtienen 64000 bits por segundo (bps). La base para la infraestructura del teléfono es 64000 bps (64 kbps).

Normalmente, se utilizan dos variaciones básicas de PCM de 64 kbps; la ley u, que es la estándar usada en EEUU, y la ley a, que es la estándar utilizada en Europa. Los dos métodos son similares en cuanto ambos utilizan la compresión logarítmica para pasar de 12 a 13 bits de calidad PCM lineal en palabras que tienen sólo 8 btis, pero se diferencian en detalles de compresión relativamente pequeños. El método de la ley u tiene una pequeña ventaja sobre el método de la ley a en términos de rendimiento de la relación señal-ruido de bajo nivel.

3.1.4. Compresión de audio

Para comprimir una señal de audio, se asigna a un rango de niveles un único valor de reconstrucción. Esto hace más fácil discernir entre niveles de amplitud, reduciendo el efecto del ruido que se pueda añadir en una transmisión o en un proceso de lectura. El precio que hay que pagar es una distorsión de la señal, ya que esta no recupera su amplitud original en todos los puntos, sino un valor próximo. Esta distorsión puede verse como ruido añadido, en una proporción que se puede controlar variando el número de niveles de cuantificación: cuantos más niveles, menos ruido.

Si en una zona del espectro se introduce ruido sin que se oiga, se realiza una cuantificación menos fina (escalones de cuantificación más grandes, que se traduce en menos bits), mientras que en las zonas donde el ruido se hace audible, se asigna más bits. Es en este punto donde se diferencian unos codificadores de otros. El cálculo de la cantidad de ruido que se puede admitir es un dato basado en lo que se llama el "Modelo psicoacústico". Este modelo es completamente experimental, y se realiza promediando la respuesta de muchas personas frente a determinados estímulos. Un buen modelo permitirá estimar con precisión la cantidad de ruido admisible y la banda donde puede introducirse con pérdidas mínimas, mientras que las estimaciones de un mal modelo no permitirán comprimir tanto o con tanta calidad. No obstante, la elección de un modelo u otro puede venir determinada por diferentes factores, como por ejemplo, el coste computacional.

El procedimiento básico es el siguiente:

1. Crear ventanas con la señal: tomar muestras durante unos 10 ms (alrededor de 512 muestras). A este intervalo temporal se le denomina ventana de análisis.

2. Análisis espectral de la ventana: se divide la señal en subbandas, generalmente unas 32, que suelen distribuirse de manera uniforme en frecuencia. Hay que calcular un umbral de enmascaramiento para cada una de estas bandas.

3. Generalmente se usa una *TF* (Transformada de Fourier) o mejor una *FFT* (Transformada rápida de Fourier), pero pueden utilizarse otras transformaciones, como por ejemplo la *DCT* (Transformada Discreta del Coseno). Al aplicar esta transformación, el espectro se divide en bandas de anchura creciente con la frecuencia, lo que simula el comportamiento del oído, que tiene más resolución espectral en baja frecuencia.

4. Cálculo de los umbrales de enmascaramiento: esta parte puede hacerse de dos formas. La más simple, y la que se usa para factores de compresión pequeños, es utilizar la energía de las subbandas para estimar los umbrales, que es computacionalmente

poco costoso. Para elevar los factores de compresión, se necesita afinar más en la estimación, lo que se hace calculando una FFT de muchos puntos (más de 512) o calcular FFT (o MDCT) de cada una de las subbandas. La decisión de usar uno u otro método es un compromiso entre prestaciones y coste computacional.

5. Cuantificación: según los umbrales de enmascaramiento y la velocidad binaria de salida se realiza la cuantificación de los coeficientes de cada banda con un número determinado de bits.

Y este es el proceso básico. Evidentemente una implementación real debe tener en cuenta muchos otros factores. También suele ser frecuente aplicar un algoritmo de compresión estándar a la salida o algún esquema de compactación eficiente de bits para reducir aún más la tasa binaria.

3.1.5. Otras consideraciones

A continuación se tratan otras consideraciones a tener en cuenta con el tratamiento del audio digital.

Detección de voz. VAD

Existen muchas y muy complejas técnicas para detectar la voz. Aquí se va a mostrar la técnica mas básica de todas. Consiste, básicamente, en calcular la media ponderada de la señal cuando no hay voz, para más tarde comparar ese valor promedio con los obtenidos en las muestras actuales. Si superan ese valor se considerará que la señal es válida -hay voz- en caso contrario se ignorará la muestra. Hay dos formas para que se pueden usar para este menester:

Cálculo de la magnitud. La magnitud se define como la suma del valor absoluto de las muestras que se encuentran en una ventana; se divide por el número de muestras para normalizar los resultados:

$$M(i) = \frac{1}{T_v} \sum_{k=0}^{T_v-1} \mid m(k) \mid \tag{3.4}$$

siendo $M(i)$ la magnitud de la ventana i, T_v el tamaño de la ventana y $m(k)$ la muestra de sonido capturada.

Cálculo de la energía. Se define como la suma de las muestras al cuadrado; se divide

por el número de muestras para normalizar los resultados.

$$M(i) = \frac{1}{T_v} \sum_{k=0}^{T_v-1} m^2(k) \tag{3.5}$$

siendo $E(i)$ la energía de la ventana i.

Modificación del volumen en el audio digital

Para modificar el volumen, se multiplican todas las muestras por un determinado factor. si el factor es mayor que uno, se incrementa el volumen, si es menor que uno, se decrementa y si es uno el volumen no varía.

Hay que tener en cuenta que, si al multiplicar una muestra por un factor mayor que uno se produce desbordamiento se truncarán los dígitos superiores, pero, esa solución no es adecuada; lo correcto es dejar la muestra con el mayor valor posible. Esto lo se debe implementar mediante codigo específico. Para el caso particular de pretender aumentar el volumen al máximo, sin que se produzca saturación, se debe buscar la muestra de mayor valor absoluto y calcular el factor con el que multiplicar para convertir esta muestra en el mayor valor que permita el rango. Posteriormente hay que multiplicar todas las muestras por ese factor.

Mezcla de sonidos digitales

Basta con sumar las muestras de forma ordenada. Sólo hay que tener en cuenta que no se produzca truncamiento (*clipping*). Para ello, antes de realizar al suma, se multiplica cada sonido por un factor menor que la unidad. Con esta regla se evita la posibilidad de que se produzca un truncamiento, pero en general, el volumen del sonido final es muy bajo, se debe reajustar para que ronde el 90 % del rango permitido.

El factor óptimo -menor que uno- con el que se deben multiplicar todas las muestras a mezclar es el inverso del número de fuentes a mezclar, es decir $f = \frac{1}{N_f}$, donde N_f es el número de fuentes.

3.2. Protocolo de Internet. IP

Hoy por hoy, el Protocolo de Internet -*IP*, (Internet Protocol) es el protocolo más usado en las redes de comunicaciones telemáticas, por esta razón se ha elegido para el desarrollo del proyecto **Asubío**. A continuación se expone una breve introdución en el fun-

cionamineto de *IP*, resaltando aquellas carácterísticas que pueden afectar al rendimiento de ésta aplicación.

3.2.1. Introducción

En general una red de comunicaciones se compone de diferentes medios. Es necesario que se garantice la comunicación entre sí de todos los equipos conectados a la red. El nivel de red oculta los detalles del medio, ofreciendo al usuario la imagen de una única red, aunque dicha red esté formada por varias redes distintas en medios físicos diferentes. Así este nivel se encarga principalmente de la comunicación extremo a extremo y el encaminamiento. IP es un protocolo que se desenvuelve en el nivel de red, nivel 3 de *OSI* (Open Systems Interconection), su función es hacer lo mejor que pueda la entrega de un datagrama al host destino. Las funciones de *IP* son:

- Provee una red virtual única independiente de los medios físicos y protocolos de enlace.
- Cada equipo que se conecta debe tener al menos una dirección que lo identifica dentro de la red, por lo que provee de un direccionamiento lógico uniforme independiente de las redes físicas.
- Aísla a la capa de transporte de la tecnología, topología, etc. de la red.
- Se encarga del establecimiento de rutas.
- Control de congestión dentro de la red.
- Adaptación al tamaño de trama del nivel de enlace (MTU).

IP es un protocolo que proporciona servicios no orientados a conexión y no es fiable. Es un protocolo bien intencionado, es decir, un paquete se pierde solamente cuando se dan circustancias excepcionales como un corte en la red. *IP* es el principal protocolo de la capa de red. Este protocolo define la unidad básica de transferencia de datos entre el origen y el destino, atravesando toda la red. Además, el software IP es el encargado de elegir la ruta más adecuada por la que los datos serán enviados. En resumen, se trata de un sistema de entrega de datagramas que tiene las siguientes características:

- Es no orientado a conexión, cada uno de los paquetes puede seguir rutas distintas entre el origen y el destino, puediendo llegar duplicados o desordenados.
- Es no fiable, los paquetes pueden dañarse por errores en los bytes durante la transmisión por el medio, perderse o llegar retrasados por un culpa encaminador congestionado, etc.

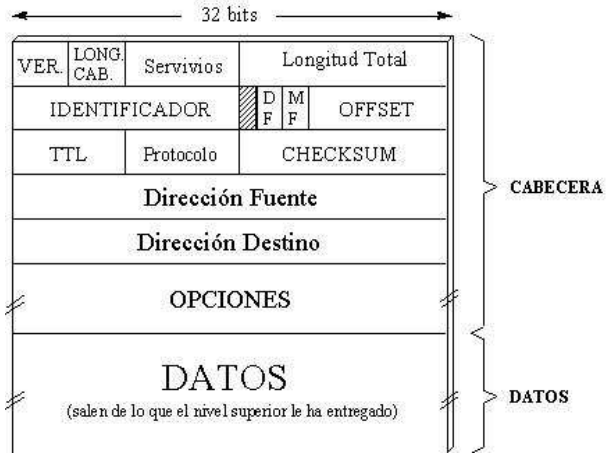

Figura 3.5: Cabecera de un paquete IP

- Cada datagrama es tratado en la red de manera independiente de los demás.
- El tamaño máximo del datagrama es de 65635 bytes, pero se adapta al nivel de enlace.

IP es un protocolo adaptativo, es decir, en todo momento se realiza una comprobación de la mejor ruta a seguir para el siguiente salto comprobando la tabla de encaminamiento del nodo actual, las entradas de la tabla de encaminamiento pueden cambiar en cualquier momento dependiendo de las condiciones de la red. Por ejemplo si un enlace deja de funcionar se enviaran los datagramas por una ruta diferente, si es que existe. Un cambio en la topología de la red puede hacer que los datagramas se reencaminen automáticamente. El encaminamiento adaptativo es la base de la flexibilidad y la robustez de IP. También utiliza, técnicas de fragmentación y reensamblado junto con temporizadores que permiten encaminar los datagramas a través de encaminadores congestionados o pasar de redes con grandes prestaciones a redes pequeñas y de baja calidad de tráfico. Esto hace posible que un datagrama de IP viaje pasando por una gran variedad de tecnologías de comunicación que van desde las redes de telefonía básica hasta los enlaces dedicados por satélite o fibra óptica y viceversa.

El funcionamiento de IP se basa en un identificador de 32 bits (dirección IP) de un sistema (interfaz). La dirección ip está vinculada a un punto de acceso al servicio, es decir, una dirección identifica de forma única a un punto de acceso remoto con el cual comunicar. Esto es lo que posibilita el encaminamiento, la dirección es única en el contexto en que está siendo utilizada -la red-, ya que identifica perfectamente al destino. Para explicar el tema de manera superficial, IP no necesita conocer la ruta completa que lo llevara a la

red destino, sólo necesita descubrir cuál es el siguiente salto (gateway o puerta de enlace) y enviar allí el datagrama.

Todas las funciones que aseguran la fiabilidad del envío y entrega de datos se ha concentrado en la superiores a *IP* como se detallará posteriormente. Cada datagrama de IP tiene un tiempo de vida (TTL) que expira según se configure el temporizador por TCP, ésto provoca la retransmisión del datagrama por parte de TCP.

3.2.2. Rendimiento de *IP*

El rendimiento del protocolo *IP* depende de las siguientes características:

1. **Ancho de banda de la transmisión.** Depende de la Certificación de la Red y de sus parámetros.
2. **Memoria de los buffer.** Depende del software, encaminadores, y otros equipos de LAN que procesan los datagramas IP.
3. **Capacidad de procesamiento de la CPU.** Características de los servidores.

El rendimiento también se puede ver afectado por el parámetro Unidad Máxima de Transferencia, MTU (Maximum Transfer Unit). El MTU determina la longitud máxima, en bytes, que podrá tener un datagrama para ser transmitida por una red física. Obsérvese que este parámetro está determinado por la arquitectura de la red: para una red Ethernet el valor de MTU es de 1500 bytes. Dependiendo de la tecnología de la red los valores MTU pueden ir desde 128 hasta unos cuantos miles de bytes. Si el paquete IP, supera el MTU de la red física, este será fragmentado en varios paquetes, con la consiguiente perdida de tiempo y de ciclos de CPU. Estas consideraciones se deben tener en cuenta.

Estos son los puntos críticos que afectan el rendimiento de una red *IP*, no se conocen mecanismos de control. El diseño de un protocolo es una lucha constante contra entre ganancias y pérdidas de eficiencia.

3.2.3. Protocolo de Control de Transferencia. TCP

Proporciona un mecanismo fiable para la transferencia de flujos de información. Aunque está íntimamente relacionado con IP, TCP es un protocolo independiente de propósito general. Al ser un protocolo de alto nivel (4 en OSI) su función es que grandes volúmenes de información lleguen a su destino correctamente, pudiendo recobrar la pérdida esporádica de paquetes. El protocolo TCP tiene las siguientes características:

- Proporciona comunicación bidireccional completa mediante circuitos virtuales.

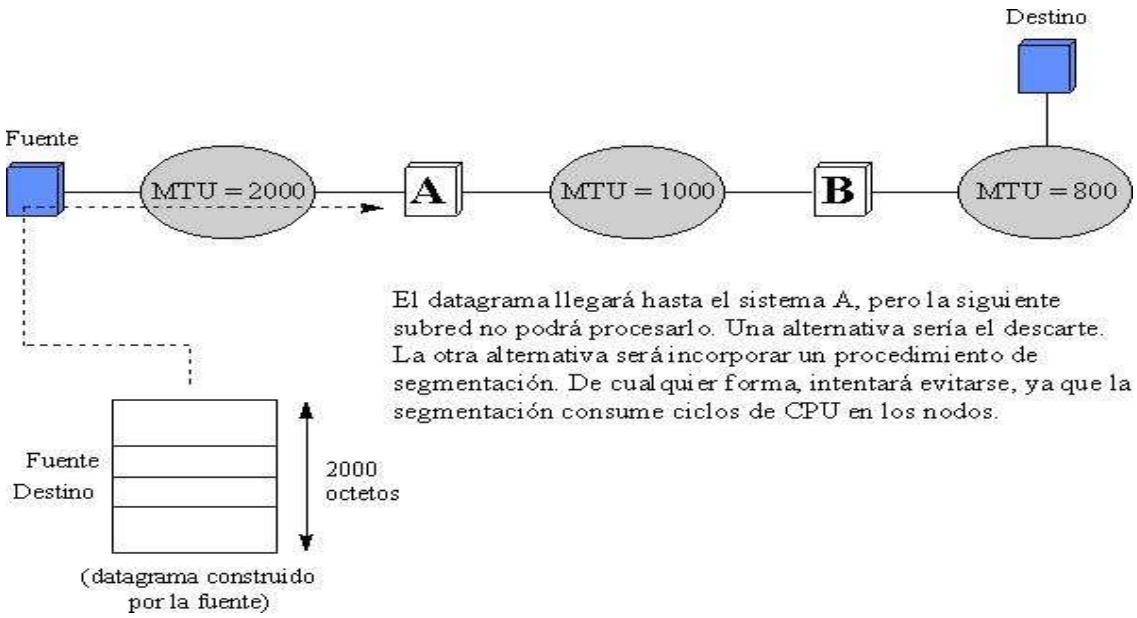

Figura 3.6: Rendimiento del protocolo IP

- Desde el punto de vista del usuario la información es transmitida por flujos de datos.
- Confiabilidad en la transmisión de datos por medio de:
  - Asignación de números de secuencia a la información segmentada.
  - Validaciones por suma (checksum).
  - Reconocimiento de paquetes recibidos.

Fiabilidad en la transferencia de TCP. Cada vez que un paquete es enviado se inicializa un contador de tiempo, al alcanzar el tiempo de expiración, sin haber recibido el reconocimiento, el paquete se reenvía. Al llegar el reconocimiento el tiempo de expiración se cancela.

A cada paquete que es enviado se le asigna un número de identificador, el equipo que lo recibe deberá enviar un reconocimiento de dicho paquete, lo que indicará que fue recibido. Si después de un tiempo dado el reconocimiento no ha sido recibido el paquete se volverá a enviar. Obsérvese que puede darse el caso en el que el reconocimiento sea el que se pierda, en este caso se reenviará un paquete repetido.

Utiliza el principio de ventana deslizable para esperar reconocimientos y reenviar información. Se define un tamaño de la ventana, que serían el número de paquetes a enviar sin esperar reconocimiento de ellos. Conforme se recibe el reconocimiento de los primeros

paquetes transmitidos la ventana avanza de posición enviando los paquetes siguientes. Los reconocimientos pueden recibirse en forma desordenada.

Si el protocolo sólo contara con reconocimientos positivos, gran parte de la capacidad de la red estaría desperdiciada, pues no se enviarían más paquetes hasta recibir el reconocimiento del último paquete enviado. El concepto de ventana deslizante hace que exista una continua transmisión de información, mejorando el desempeño de la red.

3.2.4. Protocolo de Datagramas de Usuario. UDP

Proporciona de mecanismos primordiales para que las aplicaciones se comuniquen entre si en máquinas remotas. Provee un servicio no confiable orientado a no conexión, por lo que el programa tiene toda la responsabilidad del control de confiabilidad, mensajes duplicados o perdidos, retardos y paquetes fuera de orden.

Este protocolo deja a la aplicación la resposabilidad de una transmisión fiable. Con él puede darse el caso de que los paquetes se pierdan o bien no sean reconstruídos de forma adecuada. Permite un intercambio de datagramas más directo entre aplicaciones, por esta razón, es idóneo para software que busque optimización en el ancho de banda consumido, ya que, UDP no sobrecarga la red con confirmaciones y avisos de datagramas desordenados o perdidos. Puede y debería elegirse para aquellas que no demanden una gran cantidad de datagramas para operar óptimamente, como es el caso de Voz sobre IP.

3.2.5. Unicast *versus* Multicast

Los servicios "tradicionales" de Internet están basados en lo que se conoce como IP Unicast, es decir, datagramas con un único origen y un único destino. En este caso, la comunicación se denomina "uno a uno". Sin embargo, en situaciones en las que hay más de dos participantes en la comunicación, la utilización de IP Unicast no es eficiente, puesto que hay que enviar la misma información a varios destinatarios simultáneamente, lo que puede sobrecargar a los emisores y la red.

Tradicionalmente, este problema ha sido resuelto mediante la utilización de "reflectores" (o Unidades de Control Multipunto , MCU, según la terminología *H.323*), que son equipos a los que se conectan todos los participantes y que se encargan de la replicación de todos los datagramas que envía un participante al resto de ellos. Esta solución plantea varios problemas, sobre todo referentes al consumo excesivo de ancho de banda en la red. Frente a esta solución surge la alternativa IP Multicast, que consiste en que la red se encargue de hacer llegar los datagramas enviados por el emisor a un grupo de receptores.

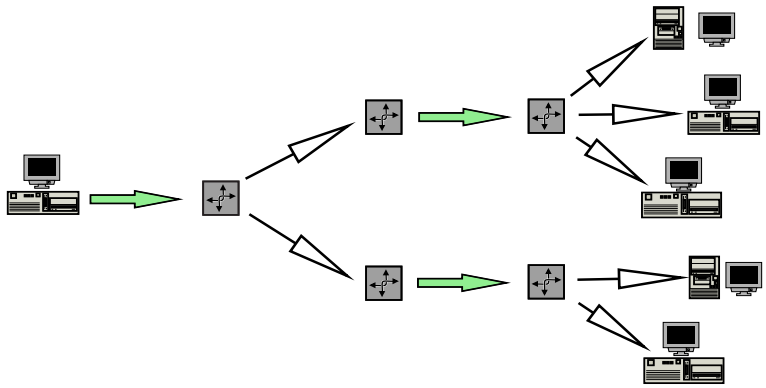

Figura 3.7: Funcionamiento de IP Multicast

La idea de grupo se implementa mediante un rango especial de direcciones IP y la suscripción o no a un determinado grupo es dinámica y la decide el receptor en función de sí está interesado o no en recibir los datagramas dirigidos a dicho grupo. Lo más importante es que la propia red se encarga de replicar los paquetesparaque lleguen a todos los suscriptores del grupo. Además, todo ello se realiza de modo que por cada enlace de la red solo circule un paquete (figura 1), duplicándose éste en aquellos momentos en los que se llegue a una bifurcación en el camino hacia los destinatarios.

Cuando inicialmente se diseño el esquema de direccionamiento en Internet se definieron varias clases de direcciones. La clase D se reservó para las direcciones multidestino, esto es, el rango que va desde la dirección 224.0.0.0 a la 239.255.255.255. Dichas direcciones son conocidas como “direcciones de grupo” o multicast. De todo este rango, ciertas direcciones están reservadas para un uso específico mientras que otras quedan libres para que las usen las aplicaciones que utilizan los servicios IP Multicast.

3.2.6. IPv6, la próxima generación IP

El protocolo IP en su versión 6 (IPv6, a partir de ahora), surge como un sucesor de la versión 4, que pronto se quedará corta de espacio de direcciones, debido al crecimiento exponencial de Internet.

Al ver en el horizonte estos problemas, el IETF comenzó a trabajar en 1990 en una versión nueva de IP, que sería más eficiente y con un rango de direcciones virtualemente inagotable, siendo además más flexible, eficiente y segura. Además, de prestar una mayor atención al tipo de servicio, para poder atender adecuadamente al tráfico multimedia que se avecina. Los cambios de IPv6 respecto de IPv4 son, de forma resumida:

- Expansión de las capacidades de direccionamiento. IPv6 incrementa el tamaño de las direcciones de 32 bits (IPv4) a 128 bits, para soportar más niveles en la jerarquía de direccionamiento, un número mayor de nodos direccionables, y un sistema de autoconfiguración de direcciones.

- Simplificación de la cabecera. Algunos campos de la cabecera del IPv4 son eliminados o pasan a ser opcionales, tanto para reducir el coste de procesamiento como el tamaño de la cabecera.

- Mayor flexibilidad para extensiones y nuevas opciones. En IPv6 no existe un campo opciones, como tal. Eliminando así las limitaciones de tamaño en la cabecera, e introduciendo una gran flexibilidad en el desarrollo de nuevas opciones.

- Capacidades de control de flujo. Se añaden capacidades que permiten marcar los paquetes que pertenezcan a un determinado tipo de tráfico, para el cual el remitente demanda una calidad mayor a la especificada por defecto o servicios en tiempo real.

- Capacidades de autenticación y privacidad de datos. IPv6 provee extensiones para soportar autenticación, e integridad y confidencialidad de datos.

Direcciones IPv6

Con un octeto se pueden representar los números de 0 a 255. Por tanto las direcciones IPv4 se componen de cuatro octetos, o 32 bits, lo cual genera más de cuatro millones de direcciones. En IPv6 las direcciones se componen de 16 octetos, es decir 128 bits. Esto daría lugar a 2 128 direcciones, más o menos 340 sextillones. No obstante, esta cifra no se alcanza, ya que parte de los dígitos identifican el tipo de dirección, con lo que se quedan en 3800 millones. En cualquier caso se garantiza que no se acabarán en un plazo razonable.

Hay tres tipos de direcciones: *unicast*, *anycast* y *multicast*. Las direcciones *unicast* identifican un sólo destino. Un paquete que se envía a una dirección unicast llega sólo al ordenador al que corresponda. En el caso de las direcciones *anycast* se trata de un conjunto de ordenadores o dispositivos, que pueden pertenecer a nodos diferentes. Si se envía un paquete a una de estas direcciones lo recibirá el ordenador más cercano de entre las rutas posibles. Las direcciones *multicast* definen un conjunto de direcciones pertenecientes también a nodos diferentes, pero ahora los paquetes llegan a todas las máquinas identificadas por esa dirección.

Conversión formato IPv4 ⇆ IPv6

Ésta es una dirección de 32 bits: 192,9,32,12 y ésta es una dirección de 128 bits: $BAF7 : 7432 : FFFF : FFFF : 7433 : 73F9 : FFAA$. Puede observarse que las direcciones IPv4 se forman con cuatro números decimales, que como ya se ha dicho van de 0 a 255. Para las direcciones IPv6 se usan números hexadecimales. Para representar una dirección IPv4 con IPv6 se ponen a cero los cuartetos a partir del bit 33: $0000 : 0000 : 0000 : 0000 : 0000 : C009 : 200C$ Esta notación puede ser simplificada como :: 192,9,32,12 para equipos que no entienden IPv6 y como :: $FFFF$: 192,9,32,12 para equipos que sí entienden IPv4.

3.2.7. RFC's de IP

Más información sobre el Protocolo Internet se encuentra en estos RFC's:

RFC 791 Definición técnica del protocolo IP.

RFC 1122 Especificación de actualizaciones y correciones efectuadas.

RFC 1812 Requisitos y estándares que deben cumplir los fabricantes de encaminadores.

RFC 1108 Opciones de seguridad.

3.3. RTP/RTCP

RTP es un acrónimo de Real Time Protocol y como su nombre indica, es un protocolo diseñado para la transmisión de datos en tiempo real. Existen otros protocolos de tiempo real, pero éste el más usado y está estandarizado. En este proyecto se ha utilizado el protocolo *RTP* para la transmisión de audio, ya que es un contenido con características intrínsecas de tiempo real. En esta seccion, hay una pequeña introducción a este protocolo, así como su aplicación al audio.

3.3.1. Introducción

RTP (*Real Time Protocol*) provee servicios de transmisión de datos con características de tiempo real, como sonido y vídeo interactivos. Estos servicios incluyen identificación del tipo de contenido (*payload*), números de secuencia, marcas de tiempo y monitorización de pérdidas.

Normalmente, las aplicaciones suelen emplear *RTP* sobre *UDP* para poder utilizar sus capacidades de multiplexación y servicios de comprobación de errores; ambos protocolos

contribuyen a la funcionalidad del protocolo de transporte. Otras razones de peso para usar UDP en vez de TCP, es que RTP se ha diseñado pensando en los servicios multicast, para lo cual la conexion TCP no es apropiada por problemas de escalabilidad. En segundo lugar, RTP se ha diseñado para datos en tiempo real, la fiabilidad del transporte no es tan importanate como la recepción en el tiempo adecuado. TCP implementa la fiabilidad de transmisión usando retransmisión de paquetes, lo cual puede ser un incoveniente, ya que si se congestiona la red, la calidad sería muy inferior y congestionaría aun mas la red intentando retransmitir paquetes perdidos. Sin embargo, *RTP* se puede utilizar con cualquier otra red o protocolo de transporte subyacente. *RTP* soporta transmisión de datos a múltiples destinos utilizando distribución por multicast si lo proporciona la red subyacente.

Este protocolo no proporciona por si mismo ningún mecanismo para asegurar que los datos llegan en el momento que deberían o cualquier otra garantía de calidad de servicio, deja esta tarea a las capas de aplicación. No garantiza que los datos llegan ni tampoco que llegan en orden, tampoco asume que la red subyacente se encarga de que los paquetes lleguen en orden. Por estas razones RTP, se implementa generalmente dentro de la aplicación, pues estos sistemas de recuperación de paquetes perdidos o de control, deben ser implementados a nivel de aplicación.

RTP incorpora marcas de tiempo específicas para cada medio transportado (timestamp), que se utilizan para eliminar jitter (intraflujo) y para sincronizar entre flujos (interflujo). Varios paquetes pueden llevar la misma marca de tiempo si pertenecen a la misma unidad de datos a nivel de aplicación, por ejemplo el mismo cuadro de vídeo. Incorpora números de secuencia para que el receptor sea capaz de reconstruir la secuencia emitida, pero los números de secuencia también se pueden utilizar para determinar la posición de un paquete sin necesidad de reproducirlos en orden. El *payload* o identificador de carga, especifica el formato de datos, así como el esquema de compresión y descompresión, con este identificador, la aplicación receptora sabe como interpretar y reproducir los datos. Los distintos payload se definen en el RFC 1890. Otra función de RTP es identificar la fuente de los datos, lo que le permite a la aplicación receptora conocer de donde vienen los datos, por ejemplo en este proyecto permite identificar quién está hablando.

RTP representa un nuevo estilo de protocolos siguiendo los principios de enmarcado a nivel de aplicación y procesamiento de capas propuesto por Clark y Tennenhouse [1]. Esto es, *RTP* está pensado para ser maleable y así proporcionar la información necesaria por una aplicación particular y a menudo estará integrado en el procesamiento de la

aplicación en lugar de estar implementado en una capa independiente. *RTP* es un marco de protocolos que intencionadamente no está completo. Al contrario que los protocolos convencionales, en los cuales las funciones adicionales pueden aparecer haciendo el protocolo más general o añadiendo mecanismo de opciones que requeriría un *parser*, *RTP* está pensado para ser adaptado a través de modificaciones y/o adiciones a las cabeceras como sea necesario. Existen otras aplicaciones a las que se puede aplicar *RTP* como el almacenamiento de datos continuos, simulación interactiva distribuida, control activo de divisas, y aplicaciones de control y medida.

RTP consta de dos partes enlazadas muy de cerca:

El protocolo de transporte de tiempo real (*RTP*) , para transportar los datos que tienen propiedades de tiempo real. Sus características se han comentado anteriormente.

El protocolo de control de *RTP* (*RTCP*). Es el encargado de monitorizar la calidad del servicio y transportar la información sobre los participantes en una sesión.

RTCP Real Time Control Protocol, es un protocolo diseñado para trabajar conjuntamente con RTP. En una sesión RTP, los participantes se envian periódicamente distintos tipos de paquetes RTCP para tener información sobre la calidad de recepción y estadísticas de paquetes RTP perdidos, información del numero de participantes (en multicast), información adicional de los participantes, entrar y abandonar sesiones, etc. A través de RTCP se consigue:

Monitorización de la calidad de servicio (QoS) y congestión de red. Constituye la función primaria de RTCP. Esta información es util a los emisores, a los receptores y a otras terceras partes interesadas (Gateways, etc.). El emisor puede ajustar su transmisión basándose en esos informes, evaluando el rendimiento de la red para distribuciones multicast.

Identificación de fuentes. Las fuentes de datos se identifican en los paquetes RTP con identificadores de 32 bits generados aleatoriamente. Estos identificadores no aportan ningun tipo de información para el usuario final. Los paquetes RTCP SDES (Source Description, descripcion de fuente) contienen información textual: nombre del participante, telefono, direccion de correo electronico, localización, etc.

Sincroniacion Intermedia. RTCP permite sincronizar fuentes de datos que procedan de distintas sesiones RTP, para ello emplea las marcas temporales de los paquetes RTP.

Escalado de la información de control. Los paquetes RTCP se envian periodicamente entre los participantes, si el numero de participantes es grande, es necesario un compromiso entre la información actualizada y la sobrecarga de red. RTCP debe limitar el tráfico de control en la red acorde a la sobrecarga.

El protocolo *RTP*, no tiene porque ir acompañado de RTCP, ya que su unica mision es transportar los datos. De hecho, muchas aplicaciones de VoIP ni siquiera soportan RTCP, ya que no es vital para la comunicación. Esta funcionalidad podría ser asumida parcial o totalmente por un protocolo de control de sesión separado.

3.4. Negociación de la sesión

Hasta el momento, en apartados anteriores, se ha visto una serie de protocolos que se usan para transmitir los datos en tiempo real, pero para transmitir esos datos, antes es necesario negociar el inicio de la sesión que se quiere establecer. Esto es, definir o negociar una serie de parámetos, como pueden ser: los puertos de envio y recepcion de RTP, los codecs de audio empleados (payload), numero de participantes (redes multicast) etc.

Actualmente existen dos protocolos que se disputan este campo: H.323 y SIP. H.323 fue el primero en ser estandarizado por la ITU-T y usa una aproximación más tradicional, basada en el protocolo Q.931 de la RDSI y en los estándares H.XXX anteriores. H.323 no se puede considerar un protocolo abierto (el estandar no está liberado), pero ha sido la referencia de la industria hasta el momento. SIP, creado y estandarizado por IETF, -con apenas año y medio de existencia, y que todavía se encuentra en estado de desarrollo y revisión; la primera versión del RFC 2543 apareció en la primavera del 2001- se basa en un enfoque más ligero, del estilo de los protocolos de Internet, similar a HTTP (de hecho rehusa gran parte de los campos de cabecera, reglas de codificación, códigos de error y mecanismos de autenticación).

3.4.1. SIP *versus* H.323

En esta parte se compara SIP y H.323. Ambos protocolos pueden usar y usan generalmente RTP para intercambiar los datos, por tanto la elección de uno u otro no afecta a la calidad de servicio. Para realizar la comparación entre H.323 y SIP se analizaran cuatro campos: complejidad, extensibilidad, escalabilidad y disponibilidad de servicios.

Complejidad. H.323 es el más complejo de los dos protocolos, su RFC suma alrededor de 700 páginas. En cambio, SIP, con las extensiones de control de llamadas y protocolos de descripción de sesión, tiene un RFC de 260 páginas. H.323 define cientos de

elementos, mientras SIP tiene sólo 37 cabeceras, cada una con un pequeño número de valores y parámetros. H.323 usa una representación binaria para sus mensajes, basados en ASN.1 y las reglas de codificación de paquetes. SIP codifica sus mensajes como texto, similar a http y rtsp. Otra ventaja de SIP es que usa una solicitud simple que contiene toda la información necesaria, mientras muchos de los servicios H.323 requieren interacción entre los diferentes componentes del protocolo que son incluidos en el estándar.

Extensibilidad. Es una prueba para medir un protocolo de señalización de VoIP. Como con algún servicio usado excesivamente, las características de éste proveen evolución en el tiempo y como nuevas aplicaciones son desarrolladas. Esto hace compatibles las diferentes versiones. SIP se ha desarrollado teniendo en cuenta las lecciones de HTTP y SMTP (Simple Mail Transfer Protocol), ambas evolucionan con el tiempo y constituyen un rico conjunto de extensibilidad y compatibilidad de funciones. H.323 también define mecanismos de extensibilidad, pero de forma más compleja.

Escalabilidad. La escalabilidad es importante en el uso de Internet, ya que sus servicios tienden a crecer. En este caso, los protocolos deben ser comparados en diferentes niveles:

- *Números largos de dominios:* como H.323 fue creado originalmente pensado para usarse en una simple LAN, tiene algunos problemas con la escalabilidad, ni siquiera la versión más reciente define el concepto de zonas y procedimientos para usar la localización de usuarios. SIP, sin embargo, usa un método de detección de bucles verificando la historia del mensaje.
- *Tamaño de conferencias:* H.323 tiene soporte para multiconferencias con distribución de datos (multicast), esto requiere un punto central de control, llamado Multipoint Controller (MC), para procesar y señalizar, lo cual, tiende a convertirse en un embotellamiento para conferencias grandes. SIP no tiene necesidad de un MC central y la coordinación de conferencia es completamente distribuida, mejorando la escalabilidad y complejidad, pero, ofrece un proveedor de servicios con menos control.

Servicios. De un modo general SIP y H.323 proveen los mismos servicios, de igual manera nuevos servicios pueden ser incorporados. En conclusión, para servicios de control de llamadas, ambos SIP y H.323 proveen capacidades de servicios de intercambio.

En extensibilidad SIP proporciona mayores recursos para nuevas implementaciones y compatibilidad con nuevas versiones. H.323 soporta menos mecanismos debido al uso de ASN.1 que restringe los campos sujetos a nuevas implementaciones. Existen numerosas

propuestas para ampliar el protocolo SIP que todavía no han sido publicadas como RFCs ni se han añadido a SIP. Ejemplos son: el soporte para las primitivas MESSAGE, REFER ó INFO, el modo de operación requerido para atravesar NATs ó las funcionalidades añadidas de control de llamada. En escalabilidad SIP es totalmente distribuido con lo cual puede soportar conferencias de muchos participantes. En cambio H.323 presenta un esquema centralizado que reduce escalabilidad aunque tiene mayor control.

3.4.2. SDP

El Protocolo de Descripción de Sesión (SDP) se usa para describir una sesión de multimedios dentro de una solicitud SIP, es decir, se usa para propósitos de inicio de sesión, invitación a sesión, etc. El propósito de este protocolo es llevar información sobre el flujo de medios en sesiones multimedia que permiten a las personas recibir una descripción de la sesión para participar en ella. SDP es puramente un formato para la descripción de la sesión. No incorpora un protocolo de transporte, y esta pensado para usar diferentes protocolos de transporte apropiados: SIP, RTP, correo electrónico (SMTP) que usa extensiones MIME, HTTP, etc. Este protocolo es un producto del Control de Sesión Multipartita de Multimedios (MMUSIC) grupo de trabajo de la IETF, RFC 2327.

En general, SDP debe llevar información suficiente para habilitar la unión de una sesión (con la posible excepción de claves de encriptación) y anunciar los recursos a ser usados por no-participantes que pueden necesitarlos. Una descripción SDP incluye la siguiente información:

- Nombre de la sesión y propósito.
- Tiempo en que la sesión está activa.
- Los medios que comprenden la sesión.
- Información en cómo recibir esos medios tal como direcciones, puertos y formatos.
- La información adicional como el ancho de banda a ser usado en la conferencia y contactar información para la persona responsable de la sesión establecida.

3.5. XML

XML, es el estándar de *Extensible Markup Language*. XML no es más que un conjunto de reglas para definir etiquetas semánticas que organizan un documento en diferentes partes, es decir, XML es un metalenguaje que define la sintaxis utilizada para definir otros

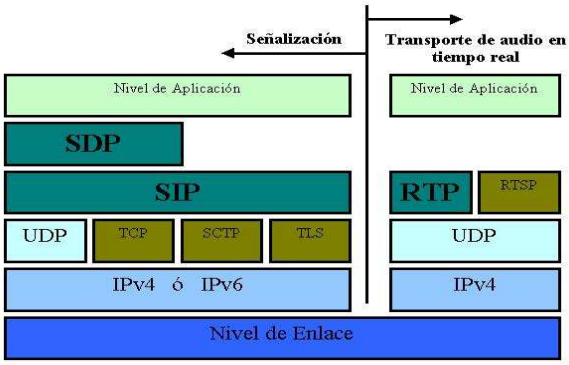

Figura 3.8: Pila de protocolos

lenguajes de etiquetas estructurados.

XML fue creado al amparo del *Word Wide Web Consortium* (W3C), organismo que vela por el desarrollo web partiendo de las amplias especificaciones de *SGML*. Su desarrollo comenzó en 1996 y la primera versión salió a la luz en febrero de 1998 bajo la definición de: "Sistema para definir validar y compartir formatos de documentos en la web".

XML tiene como objetivos:

- XML debe ser directamente utilizable sobre Internet.
- XML debe soportar una amplia variedad de aplicaciones.
- XML debe ser compatible con *SGML*.
- Debe ser fácil la escritura de programas que procesen documentos XML.
- El número de características opcionales en XML debe ser absolutamente mínima, idealmente cero.
- Los documentos XML deben ser legibles por humanos y razonablemente claros.
- El diseño de XML debe ser preparado rápidamente.
- El diseño de XML debe ser formal y conciso.
- Los documentos XML deben ser fácilmente creables.
- La concisión en las marcas XML es de mínima importancia.

Principales características

- Es una arquitectura abierta y extensible. No necesita versiones para que puedan funcionar en futuros aplicaciones. Los identificadores pueden crearse de manera simple y ser adaptados en el acto por medio de un validador de documentos (parser).
- Mayor consistencia, homogeneidad y amplitud de los identificadores descriptivos del documento con XML
- Integración de los datos de las fuentes más dispares. Se podrá hacer el intercambio de documentos entre las aplicaciones tanto en el propio PC como en una red.
- Datos compuestos de múltiples aplicaciones. La extensibilidad y flexibilidad de este lenguaje permite agrupar una variedad amplia de aplicaciones, desde páginas web hasta bases de datos.
- Exportabilidad a otros formatos de publicación (papel, web, etc.). El documento maestro de la edición electrónica podría ser un documento XML que se integraría en el formato deseado de manera directa.

Estructura de XML XML consta de varias especificaciones (el propio XML sienta las bases sintácticas y el alcance de su implementación), pero las más importantes son dos:

***DTD* (Document Type Definition):** Definición del tipo de documento. Es, en general, un archivo/s que encierra una definición formal de un tipo de documento y, a la vez, especifica la estructura lógica del documento. Define tanto los elementos de un documento como sus atributos. El DTD del XML es opcional. En tareas sencillas no es necesario construir un DTD, entonces se trataría de un documento "bien formado" (well-formed) y si lleva DTD será un documento "validado" (valid).

***XSL* (eXtensible Stylesheet Language):** Define o implementa el lenguaje de estilo de los documentos escritos para XML. Este estándar está basado en el lenguaje de semántica y especificación de estilo de documento (*DSSSL*, Document Style Semantics and Specification Language, ISO/IEC 10179).

3.5.1. Estándares XML

Los estándares Unicode e ISO/IEC 10646 para caracteres, RFC 1766 para identificación de lenguajes, ISO 639 para códigos de nombres de lenguajes, e ISO 3166 para códigos de nombres de países, proporciona toda la información necesaria para entender la versión 1.0 de XML y construir programas que los procesen.

3.6. Data Encryption Standard. DES

DES es un esquema de cifrado simétrico desarrollado en 1977 por el Departamento de Comercio y la Oficina Nacional de Estándares de EEUU en colaboración con la empresa IBM, que se creó con objeto de proporcionar al público en general un algoritmo de cifrado normalizado para redes de ordenadores. Estaba basado en la aplicación de todas las teorías criptográficas existentes hasta el momento, y fué sometido a las leyes de USA.

Se basa en un sistema monoalfabético, con un algoritmo de cifrado consistente en la aplicación sucesiva de varias permutaciones y sustituciones. Inicialmente los datos a cifrar se someten a una permutación, con bloque de entrada de 64 bits (o múltiplo de 64), para posteriormente ser sometido a la acción de dos funciones principales, una función de permutación con entrada de 8 bits y otra de sustitución con entrada de 5 bits, en un proceso que consta de 16 etapas de cifrado.

En general, DES utiliza una clave simétrica de 64 bits, de los cuales 56 son usados para el cifrado, mientras que los 8 restantes son de paridad, y se usan para la detección de errores en el proceso. Como la clave efectiva es de 56 bits, son posible un total de 2 elevado a 56 = 72.057.594.037.927.936 claves posibles, es decir, unos 72.000 billones de claves, por lo que la ruptura del sistema por fuerza bruta o diccionario es súmamente improbable, aunque no imposible si se dispone de suerte y una grán potencia de cálculo.

Los principales inconvenientes que presenta DES son:

- Se considera un secreto nacional de EEUU, por lo que está protegido por leyes específicas, y no se puede comercializar ni en hardware ni en sofware fuera de ese país sin permiso específico del Departamento de Estado.

- La clave es relativamente corta, tanto que no asegura una fortaleza adecuada. Hasta ahora había resultado suficiente, y nunca había sido roto el sistema. Pero con la potencia de cálculo actual y venidera de los computadores y con el trabajo en equipo por Internet se crée que se puede violar el algoritmo, como ya ha ocurrido una vez, aunque eso sí, en un plazo de tiempo que no resultó peligroso para la información cifrada.

- No permite longitud de clave variable, con lo que sus posibilidades de configuración son muy limitadas, además de permitirse con ello la creación de limitaciones legales.

- La seguridad del sistema se ve reducida considerablemente si se conoce un número

suficiente textos elegidos, ya que existe un sistema matemático, llamado *Criptoanálisis Diferencial*, que puede en ese caso romper el sistema en 2 elevado a 47 iteraciones.

Entre sus ventajas cabe citar:

- Es el sistema más extendido del mundo, el que más máquinas usan, el más barato y el más probado.
- Es muy rápido de calcular y fácil de implementar.
- Desde su aparición nunca ha sido roto con un sistema práctico.

Capítulo 4

Análisis de requisitos

Índice General

Este es un proyecto fin de carrera y el proceso de desarrollo seguido es evolutivo, es decir, a lo largo del tiempo, el software propuesto ha ido evolucionando añadiendo y ampliando características, hasta llegar a la situación actual. Por esta razón, en este capítulo se van a detallar todos los requisitos funcionales de la versión 0.10 de **Asubío**, sin detallar el proceso de evolución realmente seguido.

El ciclo de diseño es semejante a una espiral (Figura 4.1). La elección de este tipo de diseño se debe a que: se adapta bien a la metodología procedimental, sólo hay un único desarrollador, hay unos objetivos mínimos a cumplir y es relativamente fácil ampliar carácterísticas, evaluando la viabilidad de cada una.

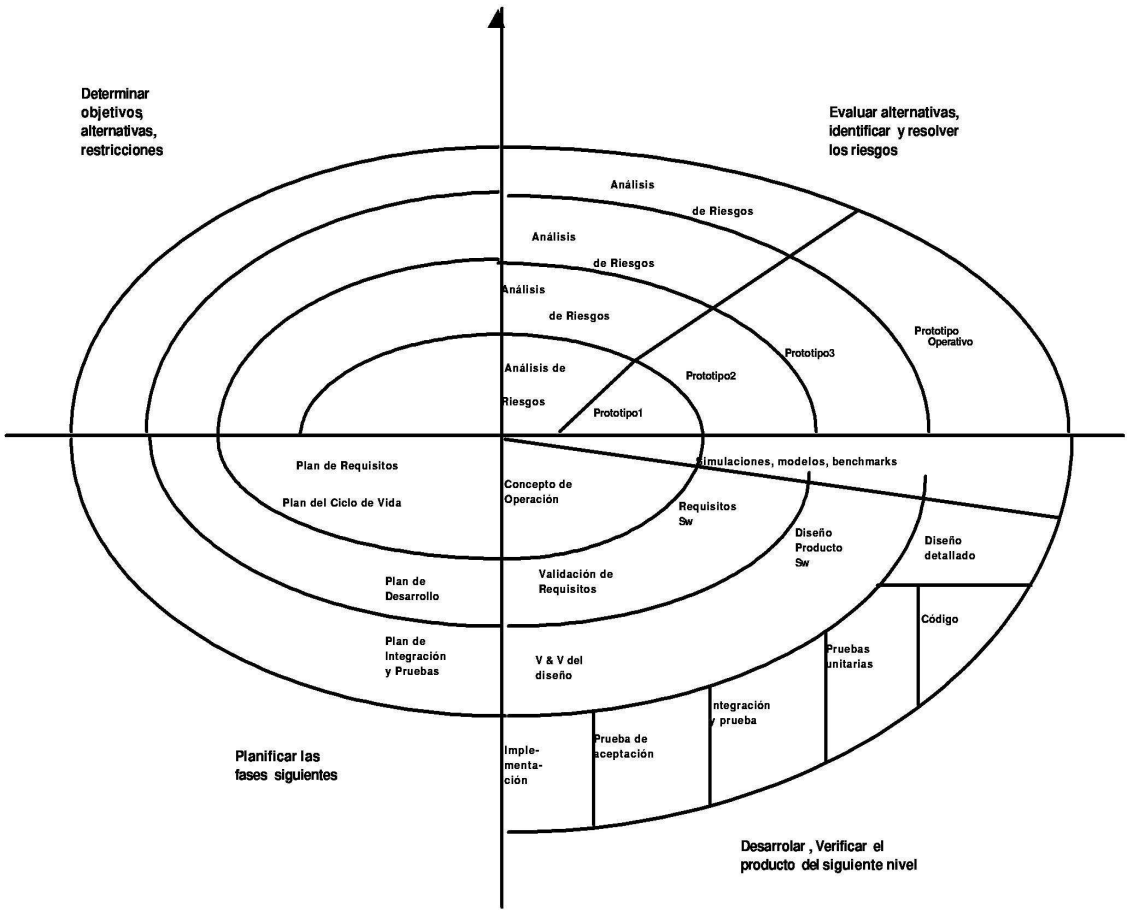

Figura 4.1: Proceso de diseño del sotware.

4.1. Consideraciones previas

El proyecto consta de dos programas complementarios, pero independientes. El programa principal se llama **Asubío** y es el software propuesto inicialmente en el proyecto. Es el encargado de transmitir el audio por canales RTP al programa remoto de otro usuario. El **Proxy SSIP** es la otra parte del proyecto -una ampliación- y principalmente realiza la función de la negociación de inicio de sesión, pero, no es necesaria su instalación, ya que fue desarrollado para casos específicos comentados más adelante. Los requisitos de ambos programas se analizarán por separado. La ilustración 4.2 muestra el uso del proxy.

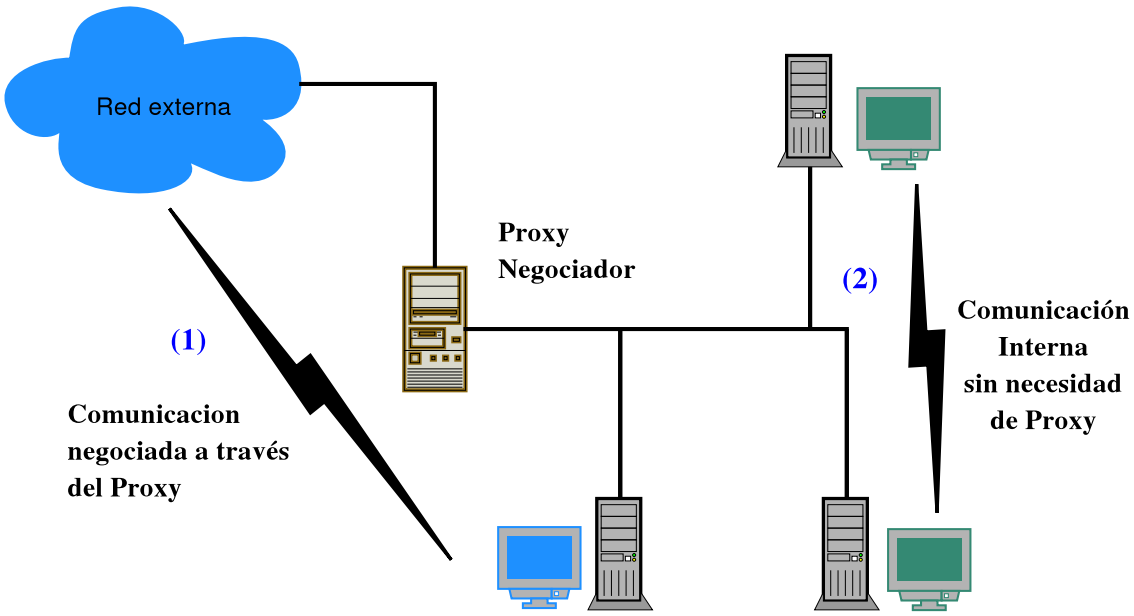

Figura 4.2: Ejemplo de uso del software.

En la situación **(1)**, se ejemplifica el uso del proxy. El proxy instalado en la máquina que hace de pasarela entre la red local y otra red (por ejemplo, Internet), ha gestionado la negociación de sesión, mientras que en la situación **(2)** el proxy no ha intervenido para nada.

Hay que señalar que en el capítulo 2 ya se ha realizado un análisis de viabilidad, se han especificado los requisitos mínimos propuestos para el proyecto y se ha explicado los motivos de la realización de este proyecto, entre otros. En este capítulo se va a detallar un completo análisis de requisitos de todo el sistema, pero partiendo del primer análisis ya realizado. Es obvio, que cualquier ampliación añadida al proyecto era viable, de otro modo no se llevaría a cabo, por lo que, no se comentaría en esta memoria.

4.2. Visión general

La complejidad filnal de todo el sistema es elevada, por lo que es necesario las especificaciones detalladas del mismo, lo que ofrecerá una guía para el diseño. Los aspectos englobados son los siguientes:

- Entorno de desarrollo.
- Entorno de operación.
- Configuración y operación.
- Arquitectura funcional del sistema.

Los dos últimos aspectos son específicos de cada programa (**Asubío** y **Proxy SSIP**), serán tratados dentro de sendas secciones posteriormente. Además de estas especificaciones genéricas, se incluyen las especificaciones funcionales en las que se detallan los comportamientos y características de los diferentes módulos necesarios para la realización del proyecto.

4.2.1. Entorno de desarrollo

Para el desarrollo de este proyecto es necesario un entorno constituido por dos máquinas PC conectadas a través de una red de área local. Aunque para las etapas de análisis, diseño y prueba, basta con una, sólo es necesario el segundo PC para las pruebas de software.

Ambas máquinas deberán tener Linux instalado con un kernel 2.4 o superior, sistema gráfico operativo (XFree86), una tarjeta de sonido compatible, soporte de dicha tarjeta en el kernel y unos altavoces -o auriculares- y un micrófono para acceder a ella. Además, deberá contar con el entorno de desarrollo apropiado para C, con su correspondiente compilador y depurador y una serie de librerías que se detallarán posteriormente. Por último, deberá disponer de toda la documentación necesaria referente al manejo de hilos, acceso a drivers de audio mediante OSS, uso de las librerías Glib y GTK+, los estándares de RTP/RTCP, etc.

A modo de resumen, el hardware, software y documentación necesaria son:

Hardware:

- Dos PC's.

- Tarjeta de red y red *IP* de área local.
- Tarjeta de sonido, altavoces y micrófono.
- Acceso a Internet (para obtener documentación).

Software:

- Sistema operativo GNU/Linux con kernel 2.4 o superior.
- Kernel configurado que soporte la tarjeta de sonido y la LAN.
- Compiladores y herramientas GNU para C: gcc, electric-fence, memprof y make.
- Librerías Glib y GTK+ junto con sus cabeceras de C ("includes").
- Herramientas de mantenimiento: diff y patch.
- Analizador de red software: tcpdump y ethereal.
- Editor de código fuente: FTE.
- Editor de interfaces gráficas de GTK+2.0: glade-2.
- Entorno de desarrollo integrado: KDevelop.

Documentación:

- Estándares para SIP y RTP/RTCP. Información de H.323.
- Documentación sobre el uso de GLib y GTK+
- Documentación sobre el acceso a drivers OSS.
- Páginas del manual de Linux.

4.2.2. Entorno de operación

El entorno de operación es el entorno en el que va a ser ejecutado el software, centrándose en lo necesario para desenvolver un funcionamiento normal. El software se puede encontrar en *http://asubio.sourceforge.net*, en forma de tarball comprimido. Una vez descargado, habrá que configurarlo, compilarlo e instalarlo, usando la herramienta *make*. Para que la compilación finalice correctamente es necesario tener instaladas en el sistema estas librerías: GLib y GTK+ (version >= 2.0) y libxml-2.

Además de todo esto, será imprescindible una tarjeta de sonido debidamente configurada y accesible. Para poder operar correctamente, es necesario una red IP, y si dentro de esa red hay un grupo de usuarios que pretenden mantener conversaciones con otros usuarios que no pertenecen a su red (ámbito), será necesario la instalación del **Proxy SSIP** que se encargue de negociar las sesiones externas.

4.3. Asubío

Análisis del programa **Asubío**.

4.3.1. Arquitectura funcional

La arquitectura funcional del programa **Asubío** consta de las siguientes áreas funcionales:

Interfaz Gráfica Usuario. GUI

Será la vía de acceso de los usuarios al sistema. Se realizará como una aplicación de fondo, minimalista, por lo que debe ser pequeña y completa, debe dar la posibilidad de ejecutar diversas acciones. Cada una de estas acciones desembocará en una sucesión de cajas de diálogo que irán preguntando al usuario por todos aquellos valores requeridos.

Deberá contener servicios de valor añadido, como por ejemplo la posibilidad de usar una libreta de direcciones -agenda de contactos- para almacenar las direcciones de otros usuarios. También sería interesante disponer de manejadores para el volumen del audio, etc.

Acceso, tratamiento y transmisión de audio

Es quizás la parte compleja de **Asubío**, porque representa el núcleo que implementa todas las operaciones con el sonido y la negociación de sesiones. Debe ser capaz de gestionar múltiples sesiones de forma simultánea, mediante el uso de hilos de ejecución paralelos.

Esta parte debe estar formada por varios subsistemas que permitan dividir el desarrollo y simplificarlo. Además la aquitectura del software, será totalmente abierta, habrá tres tipos de "plugins" que permitirán ampliar las funcionalidades:

Plugins de compresión de sonido. Codec_Plugin. También llamados "codecs" de audio ó "vocodecs". Cada *codec* implementará un algoritmo de compresión-descompresión

de audio, de esa forma, es muy fácil incorporar nuevos compresores de audio sin modificar el programa.

Plugins de Entrada-Salida de audio. InOut_Plugin. Este tipo de plugin, será encargado de proveer las funiones de lectura y escritura de audio en el PC. Principalmente accederán a la tarjeta de sonido para capturar y reporducir el sonido. También deben contener las funciones que controlen el volumen y ganancia de los altavoces y micrófono.

Plugins de Effectos generales. Effect_Plugin. Permiten desarrollar plugins para aumentar las funcionalidades del sistema original, podrán por ejemplo grabar conversaciones, modificar la voz del usuario, suprimir efectos de eco, etc.

Otras consideraciones a tener en cuenta son:

- Cada sesión RTP podrá tener plugins de codecs y de efectos diferentes a otras y además configurados de distinto modo.

- La posibilidad de cifrar sesiones con algún algoritmo de cifrado. Se debe usar compilación condicional para crear varias versiones del programa. Así se podrá exportar a países como USA, etc.

- Posicionaiento de los bytes. Linux "corre" en muchas arquitecturas diferentes. Existen arquitecturas con ordenamiento de bytes *BIG_ENDIAN*, los más significativos primero, como el PowerPC de Motorola (Apple) y otras con ordenamiento *LITTLE_ENDIAN*, los bytes menos significativos primero como ocurre en I386 de Intel. El programa deberá solucionar estos inconvenientes, para que la aplicación sea tan portable como lo es Linux.

Gestión de la Agenda de contactos

El programa dispondrá de una agenda de contactos, para guardar a los usuarios. Será un fichero en formato XML, con el siguiente DTD:

```
<!DOCTYPE agenda [
<!ELEMENT contacto (nombre[1], usuario[1],
        host[1], puerto[1],
            descripcion[1], foto[1])
<!ELEMENT contacto (default*, ignore[1])>
<!ATTLIST contacto ignore (TRUE | FALSE) #REQUIRED>
<!ATTLIST contacto default (TRUE | FALSE) #IMPLIED>
<!ELEMENT nombre (#PCDATA)>
<!ELEMENT usuario (#PCDATA)>
<!ELEMENT host (#PCDATA)>
<!ELEMENT puerto (#PCDATA)>
<!ELEMENT descripcion (#PCDATA)>
<!ELEMENT foto (#PCDATA)>
]>
```

El usuario podrá modificar todos estos campos desde una interfaz gráfica, sin necesidad de editar manualmente el fichero XML. También podrá añadir nuevos usuarios, borrar uno existente, etc. Este fichero debe contener como mínimo un contacto, el que representa a todos los usuarios que no están en la agenda. Ese contacto define el comportamiento -aceptar o rechazar automáticamente las llamadas y la foto- con los usuarios desconocidos (los que no están en la agenda) y nunca podrá ser borrado.

< **contacto** > representa un contacto de la agenda. Podrá tener dos atributos:

- **default**: si aparece este atributo y tiene el valor de *TRUE* significa que es el usuario por defecto. Siempre debe existir un usuario por defecto en el fichero para que se considere válido.
- **ignore**: si aparece este atributo y tiene el valor de *TRUE* significa que todas las llamadas de ese usuario serán igonradas.

Y estará compuesto obligatoriamente por los campos que se explican a continuación:

- **nombre**: nombre de pila del usuario.
- **usuario**: nombre del usuario en el host remoto.
- **host**: máquina o dirección IP remota del usuario.
- **puerto**: número de puerto SSIP en donde conectar con el usuario.
- **descripcion**: descripción del usuario.

- **foto**: imagen en formato *jpg*, *gif* o *png* del usuario.

4.3.2. Requerimientos funcionales

A continuación se extienden los requisitos mínimos propuestos inicialmente y que se han apuntado en el capítulo 2.

Configuración

Para poder configurar **Asubío** con las direcciones IP que debe usar, los puertos en los que debe escuchar, plugins y otros parámetros de registro que puede llegar a necesitar, es fundamental disponer de un sistema rápido y flexible, que no exija repetir el mismo proceso cada vez que se ejecute el programa.

La solución a este requerimiento viene dada por un sencillo fichero de configuración, contenido un directorio reservado a tal efecto, y basado en una sintaxis simple y potente: pares etiqueta/valor. Mediante la definición de tantas etiquetas como sean necesarias se podrá configurar el programa con múltiples parámetros.

Cada plugin, también guardará todos sus parámetros de configuración en ese mismo fichero. Por ello se debe diseñar en forma de TDA que permita acceder transparentemente al fichero.

Operaciones

A continuación se definen las operaciones básicas del programa:

- **Realizar una llamada de audio:** el usuario podrá iniciar una sesión RTP. Al llamar a esta función en la interfaz gráfica, ésta realizará los procedimientos oportunos que desencadenarán un traspaso de eventos bidireccional, hasta crear una sesión RTP. El usuario tendrá acceso a la agenda de contactos para elegir a quién llamar.
- **Gestionar agenda de contactos:** Añadir. modificar y borrar datos de la agenda de contactos.
- **Recibir una llamada de audio:** Todo comienza con un paquete de negociación SSIP, se iniciará un proceso de intercambio de eventos con la GUI para notificarle la existencia de una llamada entrante en espera. La interfaz esperará las órdenes del usuario, y en función de su respuesta, la sesión continuará ó será rechazada.
- **Finalizar una llamada de audio:** mediante este procedimiento terminará una sesión RTP.

Hay varias premisas fundamentales que se deben cumplir:

- Debe tener la posibilidad de manejar múltiples sesiones de forma simultánea: Parece claro que es necesario diseñar un mecanismo para asignar como mínimo un hilo de ejecución a cada sesión. Los hilos son muy poco costosos desde el punto de vista computacional. Además, en algunos casos ciertos procedimientos pueden realizarse con la ayuda de hilos de apoyo para evitar que se conviertan en llamadas lentas y bloqueantes: como se verá más adelante, siempre interesará un comportamiento asíncrono.

- El control de todo el sistema será de forma asíncrona: hay que evitar en todo momento el uso de funciones que puedan bloquear la GUI. Frente al sistema tradicional de llamadas bloqueantes, se deberá desarrollar otro, de llamadas asíncronas o no bloqueantes: un hilo (en general, cualquier hilo) "llamará" a un procedimiento usando un evento, y el control del programa retornará automáticamente, porque la ejecución de la rutina pedida la realizará otro hilo en paralelo. Cuando éste termine, devolverá otro evento como respuesta. Evidentemente, es más complicado usar procedimientos no bloqueantes, sobre todo porque se necesita mantener un estado muy definido, que permita saber que ha terminado la realización de un trabajo y que cuando llegue cierto evento saber qué hacer con él.

- La transmisión de audio debe ir obligatoriamente por un canal RTP, usando también el protocolo RTCP, tal y como se especifica en el RFC 1889, es decir, debe cumplir el estándar, de esta forma será posible comunicarse con usuarios de otros programas.

- Minimizar el bloqueo de la tarjeta de sonido, la tarjeta de sonido sólo estará bloqueada cuando existan sesiones RTP en curso. Al aceptar o realizar una llamada, el programa deberá comprobar si puede acceder a la tarjeta de sonido.

- Existirán tres tipos de plugins dependiendo de la función que se desee aportar, el más inmediato es la capacidad de soportar diferentes formatos de audio. Por lo que el soporte para un determinado formato, sera únicamente dependiente de la existencia de la correspondiente librería de carga dinámica. Como mínimo se debe desarrollar un plugin de cada tipo, para demostrar su funcionamiento.

- Se deberá considerar el desarrollo de algún sistema que permita el funcionamiento del software tanto en redes IPv4 como IPv6. También se debería aprovechar la potencia de las redes multicast para transportar el audio más fácilmente en un entorno multisesión.

Interfaz Gráfica Usuario. GUI

La GUI debe ser intuitiva y cómoda. Cualquier tipo de inicidencia será mostrado por pantalla. Si una operación es crítica, se deberá advertir al usuario. Se consideran operaciones críticas:

- Cerrar una sesión con un usuario.
- Salir del programa.
- Cambiar el plugin de entrada-salida de audio.
- Cualquier cambio en la configuración que afecte a sesiones posteriores: cambio de puertos de negociación, cambio de proxy, etc.

Casi todas las ventanas de la aplicación deben ser no bloqueantes, excepto las siguientes:

Agenda de contactos. Debe permitir gestionar todos los usuarios.

Diálogo de conexión. Servirá para iniciar una sesión con otro usuario. Deberá permitir lanzar la ventana de gestión de agenda de contactos y elegir a la persona con que conectar, pero, también debe permitir introducir esos datos manualmente. Se podrá especificar el "codec" o "codecs" que usar para transmitir el audio. Eventualmente, para usuarios avanzados, se considerará la posibilidad de introducir otros parámetros: puertos RTP, desabilitar SSIP, etc.

Ventana de configuración. Todo el sistema de configuración debe ser íntegramente gráfico y debe permitir configurar los parámetros que afectan al rendimiento de la aplicación, así como la configuración de los distintos plugins.

Dispositivo de audio bloqueado. Bloqueo de la tarjeta de sonido del PC. Si el programa no puede usar el dispositivo de audio, el usuario será notificado inmediatamente, para que finalize el programa que está usando la tarjeta en ese momento.

Otra ventana muy importante es, la ventana de llamada entrante. El usuario deberá elegir si aceptar o no la llamada. Esta ventana debe mostrarse en primer plano y en el escritorio activo en donde se encuentre trabajando el usuario. La ventana no podrá bloquear el resto de la interfaz gráfica, pero, sólo podrá existir una ventana de este tipo, esto significa, que mientras que el usuario no acepta o rechaza la llamanda no podrán aparecer otras llamadas.

4.4. Proxy SSIP

El proxy nació de la necesidad de agrupar a varios usuarios de una red, en un único ámbito. Si todos los usuarios de la red tienen a **Asubío** funcionando no tendrán ningún problema para comunicarse entre ellos, ya que cada uno tiene distinta *IP*.

4.4.1. Objeto

El problema surge cuando hay necesidad de interconectar dos redes a través de una pasarela. Esa pasarela o "gateway" enmascara a toda la red interna -donde están los usuarios- con su dirección *IP* pública, de esa forma todos esos usuarios tendrán como *IP* pública la del gateway, no la de su máquina, vistos desde una red externa. El gateway emplea un sistema llamado *NAT* (Network Address Transtation) para habilitar la comunicación entre red interna y red externa. NAT funciona bien cuando en la red interna sólo hay aplicaciones cliente de servicios, como por ejemplo: navegadores web (clientes de HTTP), clientes de FTP, etc. Esto es debido a que el sistema NAT selecciona un puerto libre -al azar- que actúa como canalizador de la comunicación entre ambas redes, es importante recalcar que el puerto se elige al azar, no hay forma de predecir cual será asignado. Sólo funiona para aplicaciones cliente, porque estas aplicaciones son las que inician la comunicación; NAT se da cuenta de que se ha iniciado un proceso de comunicacion entre una máquina interna y otra externa, anota los puertos, y gracias a esos datos, es capaz de redireccionar los datagramas IP que luego se envían desde la IP externa a la interna, sustituyendo la IP del gateway en las cabeceras de los datagramas IP recibidos, por la IP de la máquina que inició la comunicación. Este proceso es totalmente transparente para el usuario. Sin embargo, si una máquina de la red interna no inicia una conexion y NAT recibe un paquete IP externo, ese paquete será descartado puesto que NAT no sabe a que máquina interna va dirigido realmente. Este último punto es que afecta a **Asubío**, cuando llama a un usuario que está enmascarado por un gateway.

4.4.2. Alternativas

Para este problema hay tres posibles soluciones alternativas:

- Informar a NAT que todas las conexiones al puerto SSIP del gateway se redirigan al puerto SSIP de una máquina de la red interna. Esta sólucion sólo permitiría a un único usuario usar **Asubío**, áquel que usa la máquina a la que van redirigidos los paquetes.
- Asignar a todos los usuarios de la red interna un puerto SSIP no estándar. De esta forma cada usuario tendrá un puerto distinto. El procedimiento sería el mismo que

el punto anterior, pero, en vez de usar el puerto SSIP "estándar" habría que usar otros no estándar, uno por cada usuario. Si hay muchos usuarios en la red interna esta solución no es viable.

- Construir un proxy que controle transparentemente las negociaciones del exterior. Ésta es la solución adoptada. El proxy se ejecutará en la máquina que actúa de pasarela o en otra de la red interna, pero redigiendo todas las conexiones del puerto SSIP del gateway al proxy.

4.4.3. Objetivos

Dos serán las funciones del agente proxy:

- Gestionar los usuarios a los que enmascarará. El proxy creará una base de datos donde guardará los datos de los usuarios que lo estén usando.

- Gestionar la negociación de inicio de sesión entre los usuarios enmascarados. Para contactar con esos usuarios, el proxy será el encargado de gestionar el inicio de sesion RTP/RTCP.

4.4.4. Requerimientos

A diferencia de **Asubío** este proxy no tendrá interfaz gráfica y estará orientado a un "demonio" en el sistema, es decir, se ejecutará en segundo plano, por lo que no necesitará ningún tipo de GUI -Interfaz Gráfica de Usuario-. Su trabajo será gestionar las negociaciones de los usuarios que usen el proxy. Esos usuarios no tendrán que configurar nada de este proxy, tan solo deberán conocer su *IP* el puerto en donde atiende las conexiones. El administrador del sistema será el encargado de configurarlo.

Una nota importante es que el proxy podrá tener dos *IP*, una se encargaría de gestionar los usuarios -accesible desde la red interna- y la otra de gestionar la negociación de inicio de sesión -ip pública del gateway-. Los puertos de notificación y gestión SSIP deben ser distintos, aunque estén en distintas interfaces de red. El administrador podrá interrumpir el proceso de gestión de usuarios, para evitar que se den de alta o baja otros usuarios, lo cual indica que ambos procesos serán totalmente independientes.

Al igual que ocurre con **Asubío**, se deberá considerar el desarrollo de algún sistema que permita el funcionamiento del software tanto en redes IPv4 como IPv6.

Otros requerimientos que deberá cumplir el software son:

Seguridad

Debido a su situación en la red (enlace entre dos redes), se deberá tener mucho cuidado en su diseño e implementación. Cualquier fallo podrá ser explotado por un usuario mal intencionado con distintos tipos de ataques: ataques de denegación de servicio ó DoS (Denial of Service); ataques de "buffer overflow" ó desbordamiento de buffer, etc. Por esa razón deberá guardar todos los datos posibles de las conexiones e incidencias que se produzcan durante su ejecución. De está forma se podrá seguir la pista de errores o de los posibles atacantes.

Gestión de usuarios

La gestión de los usuarios consiste en controlar el número de usuarios a los que enmascara. El proxy creará una base de datos donde guardará los datos de todos los usuarios que estén online. La base de datos debe ser accesible por el administrador en cualquier momento y el formato debe ser legible (texto ascii), no puede ser formato binario. Esto es así, para que el operador-administrador pueda borrar o modificar manualmente los parámetros de los usuarios. Será necesario definir o usar un protocolo de comunicación simple que se encarge de gestionar la comunicación de alta (online) o baja (offline) de usuarios.

Gestión de negociación de inicio de sesión

Deberá controlar y asistir la negociación de sesión para todos los usuarios a los que enmascara. Para ello, ante una llamada correcta, deberá buscar el usuario al que se va dirigido en la base de datos, si no existe o no está online, se finalizará el proceso. En otro caso, es decir, si el usuario existe y los parámetros que existen en la base de datos son correctos (el programa del usuario está en el puerto SSIP almacenado, etc.), debe enlazar a modo de proxy la conexión entre el cliente que llama con el cliente al que se dirige la llamada. Durante esta situación, no se pueden bloquear otras llamadas entre otros usuarios.

Este subsistema debe ser compatible con la negociación propuesta entre clientes **Asubío**. Para ello debe usar el protocolo SSIP, definido para ese menester, en el capítulo de diseño.

Configuración

Toda los parámetros mínimos de configuración del proxy deben ser:

Dirección IP del subsitema de gestión de usuarios : especifica la interfaz de red

en la que recibirán las notificaciones de alta o baja de usuarios (subsistema de negociación).

Número de puerto del subsitema de gestión de usuarios : especifica el puerto en que negociará el subsistema de gestión de usuarios.

Dirección IP del subsitema de negociación de inicio de sesión : especifica la interfaz de red en la que negociará el inicio de sesión para los usarios activos.

Número de puerto del subsitema de negociación de inicio de sesión : es el puerto en que negociará la negociación de inicio de sesión.

Para poder configurar el proxy con las direcciones IP que debe usar, los puertos en los que debe escuchar, los parámetros de registro que puede llegar a necesitar, etc., es fundamental disponer de un sistema rápido y flexible, que no exija repetir el mismo proceso cada vez que se "lanze" el proxy. Se podrán incorporar otros parámetros en el fichero de configuración para otros requerimientos del programa, como puede ser algún tipo de limitación en el número de usuarios, etc.

4.5. Método y tecnologías de trabajo

Debido a la naturaleza de este proyecto, se optó por usar C como lenguaje de programación. Hay que tener en mente que en este proyecto importa mucho el tiempo de proceso. El audio es un flujo continúo de datos que deben se atendido regularmente, si el proceso es muy lento, se producirían cortes que imposibilitarían la comunicación. Este es un proyecto de tiempo real, la carga de trabajo del sistema también puede influir, si el sistema tiene muchos procesos, probablemente también se producirán cortes. Se eligió C como lenguaje de desarrollo por las siguientes razones:

- Todas las llamadas al sistema están implementadas en C. No cabe duda de que en este campo C/C++ sale ganando, porque la mayor parte de las APIs de sistema están escritas en C. También se puede hacer esto desde Java y C#, pero entonces se perderían sus ventajas en portabilidad (ese código en C llamado desde Java sólo sería válido para un sistema concreto) y en seguridad, ya que se trataría de código ejecutado fuera de la máquina virtual y del CLR, respectivamente.
- Portabilidad. Característica importante tanto en Java como en C#, facilita la implantación del sistema en diferentes máquinas, haciendo posible la expansión del mismo. Sin embargo, ya se ha comentado ciertos aspectos sobre la portabilidad de las dos arquitecturas anteriores: es una característica muy deseable, pero también

utópica si lo que se busca es acceso a drivers de bajo nivel, como los relacionados con el audio. Además el código en C puede ser recompilado (usando compilación condicional) para varias arquitecturas.

- Entorno de programación disponible. No cabe duda de que un programador en Linux, dispone de muchas herramientas de desarrollo, sobre todo en C. Desda la "suite" de *GNU* gcc, hasta completos entornos gráficos como KDevelop.

- C es un lenguaje que permite una programación estructurada y que presenta un elevado rendimiento a la hora de hacer aplicaciones críticas en el tiempo, como es éste caso.

A continuación se detalla una relación de las librerías que se van a emplear en el proyecto, con sus características y junto a la justificación de su elección para este proyecto.

4.5.1. Glib, Gtk+ y libxml2

A continuación se detalla una breve introducción con las características uso estas librerías, que ayudan a razonar el por qué de su elección y uso en el proyecto.

Introducción a Glib y Gtk+

La librería Gtk fue desarrollada como parte de GIMP. GIMP es un programa de dibujo muy potente y su nombre es un acrónimo de *Graphical Image ManiPulation*. GTK significa GIMP Toolkit y fue creado para crear los elementos gráficos (widgets) de GIMP, es decir, los botones, menús, iconos, etc. Posteriormente GTK fue liberada como una librería independiente de GIMP. Más tarde apareció el proyecto GNOME, GNU Network Object Model Environment - entorno de trabajo en red orientado a objetos- , con la idea de crear un entorno libre y potente para desarrollar aplicaciones gráficas. GNOME usó GTK para su sistema gráfico y comenzó a mejorarla y a desarrollar nuevas librerías. De ese empeño surgieron GLib y GTK+, posteriormente se desarrollaron librerías como libxml2, libgda/gnome-db, gnome-print, etc.

Características

Todo el proyecto GNOME, y por extensión sus librerías estan desarrolladas en el lenguaje C, implementando un sistema de Orientación a Objetos (OO). Las principales características que aportan Glib, GTK+ y libxml2 son:

Glib Es una librería que contiene multitud de funcionalidades necesarias prácticamente en todos los programas, a la vez que incluye otras funcionalidades de más alto nivel, como es, por ejemplo, el sistema de objetos. Se podría definir Glib como una librería de ayuda para la programación en C. Sus Características más destacables son:

- Crea una capa de abstracción de las diferencias entre sistemas, lo que permite que los programas que la usen, sean portables entre distintos sistemas operativos. Actualmente soporta: MS Windows, OS/2, BeOS y sistemas Unix (Linux, FreeBSD, etc.).
- Tipos de datos portables, por ejemplo, garantiza que un entero de 4 bytes en plataformas de 32 bits, siga teniendo 4 bytes de tamaño en plataformas de 64 bits.
- Posee sus propias funciones de gestión de memoria.
- Tiene estructuras de datos (listas enlazadas, arrays, etc), que implemtan TDA (Tipos Abstractos de Datos).
- Bucle de ejecución, con eventos (E/S, alarmas, temporizadores, etc).
- Sistema de objetos, facilita la creación y manejo de objetos.
- Sistema de creación y manejo de threads (hilos de ejecución o procesos ligeros), con funciones de control (semáforos, etc.)
- Totalmente conforme al estándar ANSI C y POSIX.

Gracias al empleo de Glib, se garantiza la portabilidad, entre multitud de sistemas. Además, todas sus funciones son compatibles con POSIX. La aplicación podrá ser portable hasta para entornos MS Windows. Por ejemplo, gracias al subsistema *GModule* resulta fácil desarrollar programas con carga dinámica de módulos o *plugins*, ya que independizan totalmente el entorno de programación y de ejecución, es decir, a la hora de crear un módulo, si está en una plataforma Windows usaría DDL's, mientras en plataformas Unix, crearía librerías dinámicas de enlace; igualemnte, para cargar un módulo en sistemas Windows buscaría archivos con estensión *ddl*, mientras que en sistemas Unix, buscaría archivos con extensión *so*. Con los Threads sucede algo parecido.

GTK+ Ofrece todo lo necesario para el desarrollo de interfaces gráficas, desde los "widgets"más básicos (botones, cajas de texto, menús, ventanas, etc) hasta otros mucho más complejos y elaborados que son de gran ayuda a la hora de programar aplicaciones gráficas. GTK+ está a su vez separado en varias librerías, algunas de ellas sólo disponibles para la versión 2.0 y posteriores:

GDK , implementa el nivel más bajo de la arquitectura, es decir, las primitivas gráficas. Es una librería que forma una capa sobre la implementación gráfica real (X Window, MS Windows, Mac OS X), y es por tanto la única parte de GTK+ que tiene que ser reescrita para soportar otra plataforma/sistema operativo. Es por esta razón por que ya ha sido portada a varios entornos (X Window, MS Windows, QNX, BeOS, etc.).

gdk-pixbuf es la librería que permite el tratamiento de imágenes gráficas. Esta librería permite el tratamiento (carga, visualización, grabación) de imágenes gráficas en distintos formatos (png, gif, jpeg, etc).

Pango es la parte que se encarga de la renderización de texto, permite la representación de caracteres en distintos alfabetos (occidental, cirilico, árabe, chino, etc), permite añadirle atributos al texto (cursiva, subrayado, color de fondo, etc), etc. Supone uno de los pasos más importantes dentro del proyecto GNOME para la universalización del software libre.

ATK es una librería de clases abstractas cuyo objetivo es servir de base para el desarrollo de aplicaciones accesibles para personas con deficiencias físicas. Es un desarrollo de la empresa Sun, pues forma parte de su estrategia de inclusión de GNOME en entornos Solaris.

Libxml2 Esta librería interpreta datos en formato XML. Es una de las mejores librerías para procesar XML. Está implementada en C y, como se mencionó anteriormente, forma parte del proyecto GNOME. Puede procesar DTD's, generar árboles DOM, procesar flujos de datos en formato XML (SAX), etc. Por estas razones se utilizó en el proyecto para procesar la agenda de contactos.

4.5.2. Acesso a red

Para la implementación de las funciones de acceso a la red IP se optó por la API de Sockets (Berkeley sockets), desarrollada por la Universidad de Berkeley, en deprimento de la TLI (Transport Layer Interface, desarrollada en los laboratorios AT&T), ya que la primera es la más conocida y usada en el mundo Linux. Además permite trabajar perfectamente con IPv6 e IPv4 y con redes multicast.

4.5.3. Librería RTP/RTCP

Para la conexión RTP, se empleó una librería procedente del proyecto VAT, pero modificándola sensiblemente. Se eligió esta librería por todas estas razones:

- Soporta completamente y cumple el estándar RTP. Además, también, cumple completamente con las especificaciones del protocolo RTCP.

- Está programada en lenguaje C y tiene una implementación muy eficiente.

- Es *open source* (código libre) y esta bajo la licencia GPL.

Una vez elegida librería, fue necesario realizar una auditoría de su código, lo que provocó una serie de cambios para adaptarla a este proyecto:

- Incorporación de Glib. Los tipos de datos originales se sustituyeron por los ofrecidos por Glib, también se hizo lo propio con determinadas funciones, sustituidas por sus correspondientes en Glib, así se ganó en portabilidad entre sistemas. Se incorporó el sistema de control de errores *GError*.

- Redefinición del sistema de eventos y *callbacks*. Originalmente todas los eventos (llegada de paquetes RTP, RTCP, etc.) eran gestionados por un mismo callback, lo que provocaba que la librería no se podía paralelizar. Al añadir un segundo callback se separó la parte de RTP de la parte de gestión RTCP, de forma que se logró que la librería pueda ser usada en varios hilos de ejecución simultáneamente.

- Corrección de *bugs* y eliminación de código superfluo. Al usar el sistema de control de errores de Glib, se pudo prescindir de varias funciones. También se corrigieron errores y se completaron funciones inacabadas en las partes de iniciación y finalización del cifrado de sesión con el algoritmo DES.

- Estructuración adecuada del código. Al principio, toda la librería estaba implementada en un único fichero. Este fichero se fragmetó, estructurando las funciones por su funcionalidad en varios ficheros.

Tras la realización de estos cambios, la librería ya estubo preparada para su incorporación al proyecto.

4.5.4. Sonido

El acceso a la tarjeta de sonido en Linux, para capturar y reproducir el audio y para controlar el volumen y ganacia, se pueden realizar con dos API's. Sus caraterísticas se detallan a continuación.

Open Sound System. OSS

El Sistema Abierto de Sonido es un conjunto de drivers, de dispositivo para tarjetas de sonido y otros elementos de audio, escrito para varios sistemas de tipo UNIX y UNIX-compatibles. Las versiones actuales de OSS se ejecutan en más de una docena de sistemas operativos distintos y soportan las tarjetas de sonido más populares y la mayoría de chips de audio integrados en las placas base de los ordenadores actuales.

Las tarjetas de sonido tienen generalmente distintos dispositivos ó puertos que reproducen ó graban audio. Aunque existen grandes diferencias entre todo el hardware disponible, OSS unifica todos los puertos a un pequeño conjunto. Comentamos dos de ellos, el de audio digitalizado (DSP) y el mezclador (mixer), que son los que se aplican a este proyecto, dejando de lado otros como el sintetizador ó la interfaz MIDI.

El dispositivo de audio digitalizado (también llamado DSP ó dispositivo de conversión analógico-digital ADC/DAC) se usa para grabar y reproducir sonido digitalizado, en forma de muestras tomadas a intervalos regulares de tiempo. Como es sabido, la calidad del audio dependerá del intervalo entre muestras y del número de bits para representar cada muestra.

Sin embargo, uno de los aspectos más importantes a la hora de trabajar con OSS es el hecho de que no se puede abrir varias veces el dispositivo. Como se puede suponer, este hecho es de vital importancia para el proyecto, que deberá contar con mecanismos para grabar y reproducir al mismo tiempo, con el objetivo de dar idea de fluidez en la conversación (una conversación humana normal es full-duplex).

El dispositivo mezclador (mixer) se usa para controlar el volumen de varios puertos de entrada y salida. Además, el mixer es el encargado de seleccionar las fuentes de sonido entre el micrófono, la entrada de línea y la entrada del disco compacto (CD).

En resumen, OSS tiene como ventajas: la facilidad de su uso, el hecho de que los drivers vienen integrados en el propio código del kernel y el gran número de tarjetas soportadas.

Advanced Linux Sound Architecture. ALSA

La Arquitectura de Sonido Avanzada Linux es un proyecto desarrollado para el sistema operativo GNU/Linux bajo las licencias GPL y LGPL con los siguientes objetivos:

- Crear un conjunto de drivers de dispositivo totalmente modularizados que soporten kerneld y kmod (sistema de módulos de Linux).
- Crear la API ALSA a nivel de Kernel, que deje obsoleta la API actual, basada en OSS.
- Mantener la compatibilidad con la mayoría de los drivers OSS.

- Crear la librería ALSA en C y C++, que simplifique el proceso de creación de aplicaciones que usen este sistema para acceder a los recursos de audio.

- Crear un programa en espacio de usuario para configurar el driver de forma interactiva llamado .ALSA Manager".

Como se puede comprobar, ALSA es la evolución natural de OSS. En primer lugar porque dota de mejores servicios a los usuarios del driver, implementando de mejor forma que en OSS las características de full-duplex, y también porque, mientras que el acceso a OSS se realizaba con las funciones típicas de acceso al sistema de ficheros (open, read, write y close) e 'ioctl' para controlar el dispositivo, ALSA cuenta con una API de alto nivel, accesible desde C ó C++, usando funciones sencillas.

Tras evaluar las características, para el acesso al sonido, en este proyecto se empleó el sistema OSS, ya que, los principales problemas a los que se enfrenta ALSA son, por una parte, la escasa cantidad de tarjetas de sonido soportadas, y por otra el hecho de que los drivers no se distribuyen con el árbol de código del kernel. Es decir, es necesario parchear el kernel de Linux con el código de los drivers ALSA, recompilarlo e instalarlo, operaciones no tribiales para muchos usuarios.

4.5.5. Cifrado con DES

Para cifrar una sesión se emplea DES. El estándar de RTP, junto con otros borradores (*drafts*), lo contempla como uno de los algoritmos para cifrar RTP/RTCP. Otras características por las que ha sido elegido son: principalmente por su rápida ejecución; es uno de los algoritmos de cifrado más extendido y presenta una fiabilidad suficiente para esta aplicación.

Para el algoritmo DES, se empleó una librería creada por *Saleem L. Bhatti* en Febrero de 1993, pero parcheadas para Linux por cortesía de *Mark Handley & George Pavlou* en Agosto de 1996.

La compilación con estas librerías es condicional, ya que existen países cuya legislación impide el uso del algoritmo DES.

4.6. Proceso de negociación de inicio de sesión

Para la necogiación de inicio de sesión había dos alternativas: SIP y H.323. Sin embargo se decidió crear un protocolo llamado SSIP (Simple Session Initiation Protocol), por las siguientes razones:

- H.323 es un protocolo muy complejo y grande, además es necesario comprar el estándar (no es libre), aunque existen implementaciones abiertas. Posee limitaciones de uso con NAT, es muy poco flexible y difícil de depurar, ya que tiene un formato binario. También es muy rígido con el uso de codecs de audio.
- SIP es un protocolo muy simple, pero todavía no existen implementaciones que conjuntamente con SDP, permitar crear fácilmente un sistema de negociación de inicio de sesión, es decir un "User Agent". La contrucción de una librería que implemente un *UA*, puede constituir por sí sólo un Proyecto Fin de Carrera nuevo, por lo que se escapa de este proyecto.
- Ambos protocolos negocian la sesion usando UDP, aunque SIP puede ir sobre TCP, todavía no hay implementaciones que lo usen. El hecho de ir sobre UDP, implica un proceso más complicado de negociación.

SSIP es una mezcla de SIP y H.323. Es un protocolo binario que va sobre TCP, pero su funcionamiento es muy simple. Su principal cometido es crear un sistema que permita usar **Asubío** detrás de NAT. En el futuro, como ampliación, se podría sustituir el protocolo SSIP por SIP, implementando un agente de usuario (UA).

Las especificaciones de SSIP se detallan en el capítulo de Diseño.

4.7. Otras consideraciones

En esta sección se comentarán algunos puntos de las especificaciones del sistema que no se han abordado en apartados anteriores:

4.7.1. Licencia de distribución del código

El código fuente escrito en C que constituye este proyecto (junto con el sistema de empaquetado y las posibles imágenes e iconos adjuntos) se distribuirá bajo la licencia general pública del Proyecto GNU, denominada GPL (GNU General Public License). Se puede leer esta licencia en *http://www.gnu.org/copyleft/gpl.html*

Dicha licencia permite que cualquier persona acceda, use y modifique el código presentado, siempre manteniendo el nuevo código así generado también bajo la licencia GPL. Sin embargo, este sistema de "Copyleft" no altera la propiedad intelectual que el autor de este proyecto tiene sobre todo el código escrito directamente por él ("Copyright").

4.7.2. Distribución del proyecto

Este proyecto se encuentra disponible al público en forma de código fuente en la dirección de Internet *http://asubio.sourceforge.net* Esta dirección pertenece a una cuenta creada para este proyecto en OSDN (Open Source Development Network), una red que ofrece servicios de hosting a proyectos de software siempre que sean libres y de código abierto.

Dicha cuenta sirve de referencia para el proyecto, ya que en ella se proporcionan gran cantidad de servicios que ayudan al desarrollo distribuido y al mantenimiento del código:

- Crear de listas de correo a las que se podrán subscribir todos aquellos interesados en el desarrollo y en el uso del software.
- Distribuir el proceso de escritura del código mediante el uso de un servidor CVS que coordinará la tarea de desarrollo y la hará pública.
- Posibilidad de publicación de bugs y de distribución los parches que los solucionen.
- Control de las descargas y de las versiones del proyecto.

Capítulo 5

Diseno

Índice General

Fundamentalmente el diseño se ha basado en la medodología tradicional. Sin embargo, hay determinadas partes en las que se ha usado técnicas de *POO*, desarrollando clases de objetos e incluso herencia. A pesar de que el lenguaje de programación C no es un lenguaje diseñado para simplificar la Orientación a Objetos, es perfectamente posible seguir el paradigma de la *OO* siguiendo un conjunto de reglas de diseño. Los motivos del uso de este tipo de metodología se explicarán concisamente más adelante, pero en la mayoría de los casos ha sido para facilitar el desarrollo.

Como este proyecto consta de dos programas, diferenciados e independientes, se va a analizar su diseño por separado. En primer lugar se analiza diseño del software **Asubío** y luego se analiza el diseño del **Proxy SSIP**. Los detalles del modo de implementar subsistemas y algoritmos se detallan en el capítulo siguiente.

5.1. Software Asubío

Este programa es la parte fundamental del proyecto, es el que debe cumplir la propuesta original. Para cumplir los requisitos definidos en el capítulo anterior, se sigue un diseño con las siguientes características generales:

Sistema modular. Se puede definir *sistema software modular* como aquel que ayuda a los diseñadores a construir sistemas formados por elementos autónomos y organizadas en arquitecturas sencillas.

Principio de ocultación de información. Los módulos de un sistema deben diseñarse de modo que la información contenida en ellos sea inaccesible a todos aquellos módulos que no necesiten tal información.

Abstracción de datos. Una abstracción de datos está formada por un conjunto de objetos y un conjunto de operaciones (abstracciones funcionales que manipulan estos objetos).

API explícita. Se puede decir que una *API* es explícita si el nombre identificador de las funciones que forman parte de la *API* da al desarrollador una idea intuitiva de la función que desempeña.

GUI intuitiva. Una interfaz intuitiva es aquella que logra máxima funcionalidad con mínimas funciones de manipulación.

Sistema escalable. Se dice que un sistema es escalable si su arquitectura permite de forma sencilla ampliar prestaciones, evitando modificaciones de la base estructural.

Asubío se descompone en los siguientes subsistemas:

GUI, Interfaz Gráfica : es el subsistema que emplea el usuario para interactuar con las funcionalidades del proyecto, permite al usuario de configurar el programa y ejecutar diversas opciones.

Procesador de Audio : este subsistema se encarga de gestionar el acceso a los dispositivos de audio, tanto para escritura como lectura, junto con otras operaciones.

Buffer de Audio : subsistema que controla los "plugins" de efectos de sonido *Effect Plugin* junto con los flujos de audio.

Acceso a Red : engloba todo lo referente al protocolo *IP* y superiores (TCP, UDP).

IO : subsistema que controla todo lo referente a las sesiones *RTP/RTCP*. Es un sistema complejo que se descompondrá en otros subsistemas.

Agenda XML : es el encargado de guardar y obtener la información de la agenda de contactos en XML.

Gestión *SSIP/NOTIFY* : es el subsistema que se encarga de gestionar toda la negociación *SSIP* y notificación con el *Proxy SSIP*.

También hay que tener en cuenta, que este sistema es altamente modular, por lo que habrá que definir el funcionamiento de todos los tipos de plugins.

El diagrama 5.1 muestra como interactúan todos los subsistemas en el programa.

A continuación se analiza el diseño de cada subsistema por separado, para abordar al final la integración entre todos ellos.

5.1.1. Interfaz Gráfica de Usuario. GUI

La GUI está desarrollada íntegramente en GTK+. Contiene servicios de valor añadido, como por ejemplo la posibilidad de usar una libreta de direcciones para almacenar las direcciones de contactos. Todas las incidencias (errores, informaciones, advertencias, etc ...) son cuadros de diálogo no bloqueantes, y permiten seguir usando el resto de la aplicación, excepto estas tres ventanas:

1. La agenda de contactos.
2. Diálogo de conexión o de llamada entrante.

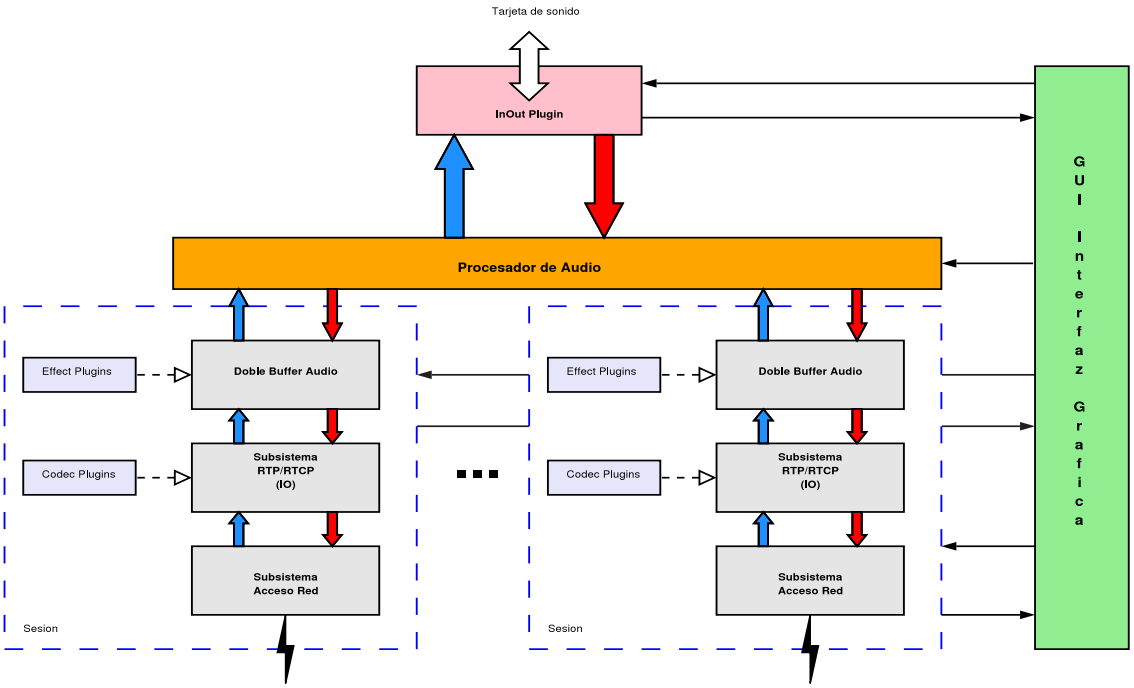

Figura 5.1: Subsistemas de **Asubío**

3. Ventana de configuración, permite configurar los parámetros que afectan al rendimiento de la aplicación, así como la configuración de los distintos plugins. Su diseño está basado en pestañas, de esa forma se permite añadir nuevas características fácilmente y de manera modular.

Otra ventana muy importante es, la ventana de aceptación de llamada entrante. El usuario puede elegir si aceptar o no la llamada. Esta ventana se muestra en primer plano y en el escritorio activo en donde se encuentre trabajando el usuario. No bloquea el resto de la interfaz gráfica, pero, sólo existe una ventana de este tipo, esto significa que, mientras que el usuario no acepta o rechaza la llamanda no podrán aparecer otras llamadas. En *OO* esto se conoce como patrón *Singlenton*.

Otro aspecto importante es el desarrollo de la interfaz principal de control de sesiones, también se basa en un sistema de pestañas, donde una pestaña representa una sesión *RTP*. Cada pestaña permite mostrar y modificar los parámetros de la sesión a la que representa. Parámetros como: control de ganancia, mute del micrófono, mostrar información de la sesión y los plugins y enmudecer al usuario.

Todo el diseño de la *GUI* es totalmente independiente de la lógica de la aplicación, siguiendo el patrón *MVC*. De esta forma será muy fácil modificar o desarrollar en el futuro

la GUI u otras GUI's con *QT*, con *NCurses* para consola (sin GUI), etc. Como este diseño tiene un alto grado de paralelismo, todas las ventanas son creadas desde el bucle principal de GTK, para ello se ha implementado un complejo sistema de semáforos que controlan los mecanismos de transferencia de datos entre varios hilos de ejecución.

5.1.2. Acceso a red

Como el sistema debe funcionar tanto en redes IPv4 como IPv6, se desarrolló un sitema que permite abstraer completamente las funciones del API de sockets. Aunque esta parte no se desarrolló usando OO, el siguiente diagrama (5.2 UML permite representar fácilmente su estructura. Además, este subsistema, debe proveer funciones para acceder a redes *multicast*

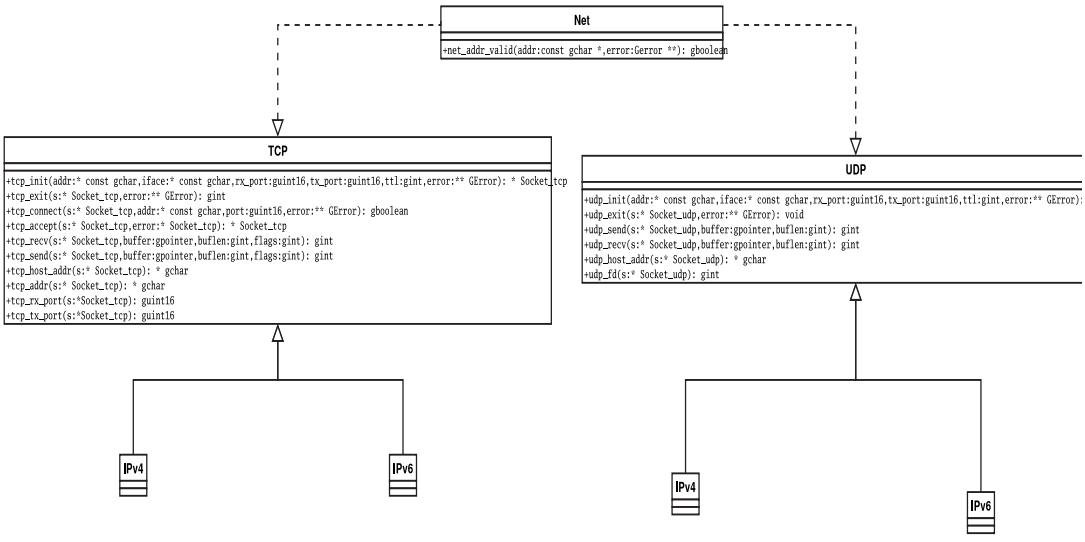

Figura 5.2: Relaciones lógicas existentes en el acceso a red

5.1.3. Acceso al fichero de configuración

Hay un TDA que se encarga del acceso a toda la configuración, carga todos los pares *clave=valor* que se encuentran en una sección *[GLOBAL]* por ejemplo. El fichero de configuración presenta el siguiente formato:

```
[GLOBAL]
rtp_base_port=10000
plugins_inout_dir=/home/riguera/.asubio/plugins/inout/
plugins_codecs_dir=/home/riguera/.asubio/plugins/codec/
plugins_effect_dir=/home/riguera/.asubio/plugins/effect/
contact_db=/home/riguera/.asubio/agenda.ml
program_hostname=192.168.1.2
audio_mic_vol=58
audio_spk_vol=65

[OSS]
audio_device_in=/dev/dsp
mixer_device_in=/dev/mixer
audio_source=0
audio_device_out=/dev/dsp
mixer_device_out=/dev/mixer
volume_control=0
```

El TDA lee el fichero en memoria y por medio de listas enlazadas que contienen los pares clave-valor de una sección, se puede mantener en memoria toda la estructura. Además esa estructura incorpora semáforos binarios que impiden que dos *threads* o hilos, accedan y modifiquen una sección al mismo tiempo. De esta forma el TDA se puede usar con hilos concurrentes.

5.1.4. Procesador de audio

Es uno de los subsistemas más crítitos del programa, sus dos tareas principales son repartir el audio de entrada (generalmente del micrófono) y mezclar el audio de salida (hacia los auriculares o altavoces). Es un sistema autónomo, es decir, actúa concurrentemente junto con otros subsistemas y es capaz de realizar las siguientes funciones:

- Repartir el audio a todas las sesiones u otros subsistemas que lo soliciten.
- Mezclar el audio de distintas fuentes, generalmente sesiones RTP.
- Controlar el tiempo de escritura y lectura de bloques de audio. Si se escribe o lee demasiado pronto o demasiado tarde en un dispositivo de audio, se producen *buffer overrun* y cortes de audio. Los *buffer overrun* provocan cortes en el audio y una pérdida de rendimiento global del programa. El *Procesador de Audio*, es capaz

de controlar el tiempo de forma muy precisa para evitar que se produzcan estas situaciones.

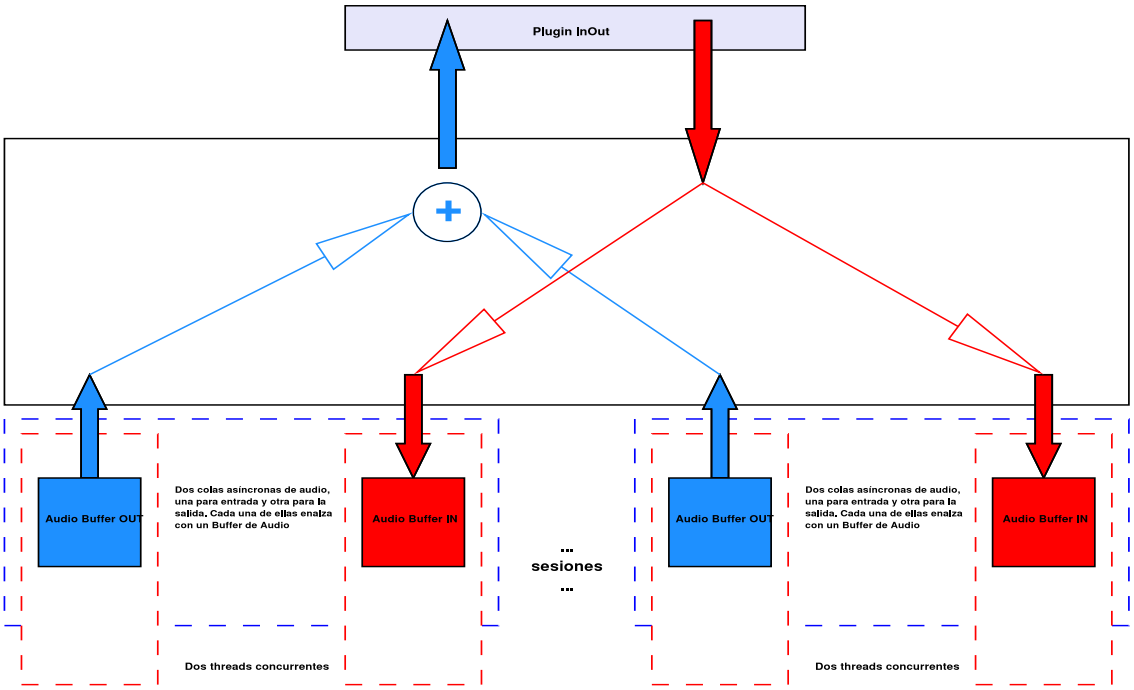

Figura 5.3: Funcionamiento lógico del Procesador de Audio

Todo el audio que se maneja en el programa es tratado por este subsistema. Dicho es un *TDA*, aunque en un contexto de patrones se asemajaría a un *Singlenton*, es decir, no pueden existir dos *Procesadores de Audio* en el programa.

El audio que despacha o recibe este subsistema, llega a través de un canal asíncrono -ya que él se encarga de gestionar el tiempo-. Ese canal tiene propiedades de cola, por lo que si una fuente envía de repente muchos bloques de audio, los bloques de audio sobrantes, serán encolados para su tratamiento posterior, es decir, actúa como un colchón frente a estas variaciones en la red. Por supuesto, dispone de un sistema de sincronización para controlar la longitud de esas colas de audio. Los canales de audio tienen también propiedades bloqueantes, es decir, si no hay bloques para leer en un canal, esa llamada quedará bloqueada hasta que lleguen datos.

Este subsistema está relacionado estrechamente con el *InOut Plugin*, ya que usa su api, en concreto sus funciones de E/S de audio para interactuar con el dispositivo. Sin embargo no tiene capacidades de inicializar el plugin, es decir, el plugin *InOut* ya está ini-

cializado cuando se ejecuta el *Procesador de Audio*.

El audio que trata este subsistema se procesa en formato 16 bits, *LITTLE ENDIAN*, mono, muestreado a 8000Hz. Si el sistema en que se ejecuta es *BIG ENDIAN* se deberá recompilar, con compilación condicional, para tratar correctamente el audio.

5.1.5. Doble buffer de audio

Este subsistema también es muy importante, cada cola de audio que se solicita al *Procesador de Audio* termina en un buffer de audio. Se puede apreciar en la figura siguiente (5.4):

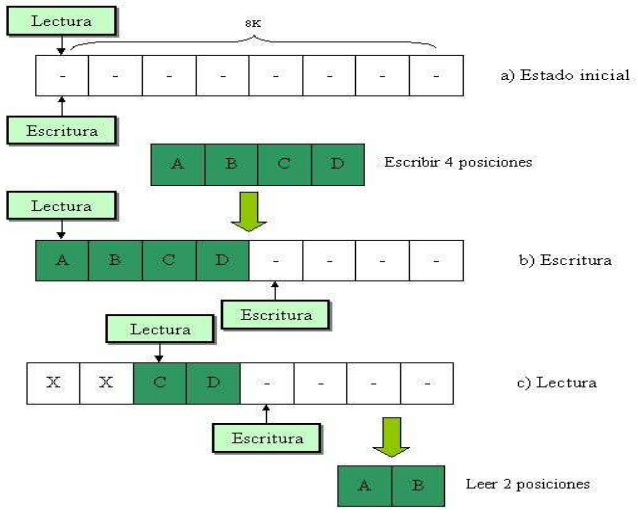

Figura 5.4: Funcionamiento del doble buffer de audio

La existencia de estos buffers de audio constituye otra parte fundamental de la arquitectura de audio. Se podría diseñar perfectamente un sistema de audio carente de dichos búfferes, consiguiendo una calidad en el sonido no necesariamente mala. Pero, gracias a los bufferes de audio, se consigue una indepencia lógica y física del *Procesador de Audio* con el subsistema RTP/RTCP, ya que están implementados por hilos diferentes que no tendrán ninguna relación en común. Se iniciarán y pararán por separado, y ninguno de los dos (en realidad son tres) tendrá constancia de la presencia del otro. Las ventajas de esta arquitectura son evidentemente, que los problemas en un hilo no afectan al otro, y que al no estar relacionados no se crearán conflictos entre ambos. El uso de hilos o procesos ligeros junto con los buffers de audio, aporta estas ventajas:

Acople de velocidades: el acople de velocidades entre la lectura/escritura del audio desde el *Procesador de Audio* y la emisión/recepción de paquetes RTP desde la red.

En un diseño basado en un sólo hilo que realiza secuencialmente una lista de tareas, el tema del acople de las velocidades puede llegar a suponer grandes dolores de cabeza por dos circunstancias: por una parte, el dispositivo de audio necesita que se le suministre las muestras de audio en una cantidad relativamente abundante (para bloquear lo menos posible el dispositivo y permitir una reproducción sin cortes. Es muchísimo mejor realizar pocos accesos y manejar más datos, que muchos pequeños accesos para pequeñas cosas) y de forma constante. Por otra parte, la red es aleatoria y no fiable, ya que son comunes los momentos de silencio en los que no llegará ningún dato para, a continuación, llegar todos de golpe.

Mejora en las posibilidades de acceso: en un sistema de sonido compuesto por un único hilo que acceda al *Procesador de Audio*, codificando/decodificando y accediendo a la red de forma cíclica y secuencial, el usuario y por ende, el sistema de audio, nunca tendrá acceso de ninguna manera a las muestras de sonido. Usando un buffer intermedio se pueden implementar ciertos sistemas, como se mostrará más adelante, para permitir por ejemplo, una lectura no destructiva del buffer.

Mayor facilidad para realizar mejoras: al dividir el problema y acordar una interfaz común (el acceso a los búfferes) es sencillo eliminar una parte y sustituirla por otra. Se puede cambiar el protocolo de transporte de los flujos (RTP) por otro, si así se estimara oportuno en cierto momento, sin tener que alterar una sola línea de las funciones que proporcionan el acceso a los recursos de audio de la máquina.

Tamaños de bloque de audio distintos el uso de los buffers permite que se puedan leer tamaños de bloque distintos del *Procesador de Audio* de los que emplean los codecs de audio.

5.1.6. Plugins

La adición de funciones en tiempo de ejecución elimina la necesidad de modificar el código base. La incorporación de los plugins permite mediante un sistema abierto crear componentes reusables del que pueden ser beneficiarias varias aplicaciones a la vez. Además, los plugins, al ser independientes del programa, permiten incorporar una serie de tecnologías propietarias o patentadas que de otra forma sería imposible.

La carga de plugins es totalmente independiente del sistema sobre el que se ejecute el programa. Glib tiene funciones que permiten abstraer completamente la creación y el uso de plugins en el software. Todos los tipos de plugins deben tener en común una única función, la que exporte una estructura al programa principal que permita acceder luego a las funciones y símbolos del plugin. De ese modo se evitan que las funciones desperdigadas.

InOut **Plugins**

Su función principal es capturar y escribir el audio en el dispositivo de sonido del PC, es decir, tal como su nombre indica, es el encargado de la entrada-salida de audio en el sistema local. La interfaz de este plugin debe aportar funciones de lectura y escritura de audio, en un dispositivo de audio. Se debe tener en cuenta que el acceso a las funciones de lectura y escritura de audio puede ser full-duplex.

Effect **Plugins**

Tienen como función expandir el funcionamiento del programa. Estos plugins pueden modificar los flujos de audio de entrada y salida para poder añadir funcionalidades como.

Codec **Plugins**

Son la clave de funcionamiento del programa, su funcion es aportar una método de compresión de audio. Estos se implementaron de forma diferente a los anteriores, en este caso existe un objeto llamado *codec* del que heredan todos estos plugins. La razones por la que se implementaron de esta forma son:

- Facilidad de programación, una vez "instanciado" el objeto, basta con llamarlo para los frames de audio de un flujo.
- Al emplear la herencia, siempre se usa la misma función, es independiente del nombre del plugin, de su estado, etc.
- Futuras ampliaciones del proyecto, como por ejemplo la posibilidad de recomprimir audio para asuntos de recuperación o retransmisión.

El siguiente diagrama UML modeliza la clase *Codec* (figura 5.5):

5.2. Concepto de sesión

Una sesión es simplemente el estado de una comunicación RTP. Para que el programa sea multisesión, aparte de lo ya explicado del subsistema de audio, del GUI, etc. necesita guardar el estado de todas sus sesiones activas. Se implementó, con ayuda de los TDA de Glib, un sistema de lista enlazada que contiene todo lo referente a cada sesión. Cada vez que se crea o destruye una sesión, se añade o borra un estado a la lista. Para obtener cualquier información de una sesión, basta con buscar esa sesión en la lista y obtener los datos de los campos significativos.

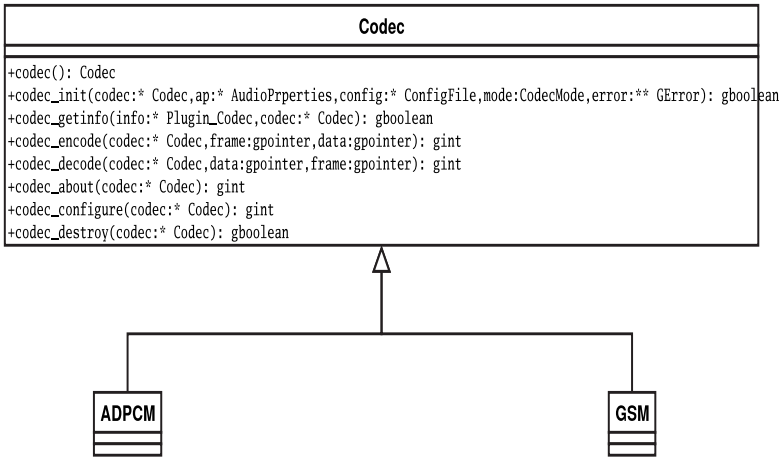

Figura 5.5: Diagrama de la clase Codec

Cuando se crea una sesión nueva, lo primero que se hace es comprobar si está lanzado el *Procesador de Audio*, o dicho de otro modo, comprobar si existen otras sesiones activas. Si no se está ejecutado el *Procesador de Audio*, se lanza. Si no funciona o está ocupado el dispositivo de E/S de audio, -*InOut Plugin*-, se muestra el mesaje de error al usuario. Después se solicita una cola asíncrona para lectura y otra para escritura del audio al *Procesador de Audio*, se aguarda la concesión y se continúa con la creación del proceso RTP/RTCP. Una vez finalizado todo el proceso lógico correctamente, se crean todos los "widgets" en la GUI. Finalmente se añade la refencia a esa sesión a la lista de sesiones activas.

5.2.1. Subsistema RTP/RTCP

Como se puede apreciar en la figura 5.6, este subsitema comprende tres hilos de ejecución. El uno se encarga de recibir los datos de la red, otro de enviarlos y un tercero se encarga de controlar a ambos y de controlar la sesión RTCP. Esta forma de distribución de los hilos permite aprovechar al máximo el tiempo de CPU concedido. Además, este diseño, favorece el cumplimiento del estándar RTP/RTCP, ya que, de esta forma el hilo RTCP se puede dedicar a controlar los timeouts de los usuarios.

Hay que tener en cuenta que estos tres hilos acceden concurrentemente a la librería de RTP/RTCP, por lo que hay que controlar esta situación con semáforos.

5.3. Simple Session Initiation Protocol. SSIP

Es el protocolo encargado de la negociación de inicio de sesión. El propósito es disponer de un protocolo que controle el inicio de sesión entre dos usuarios. El funcionamiento es

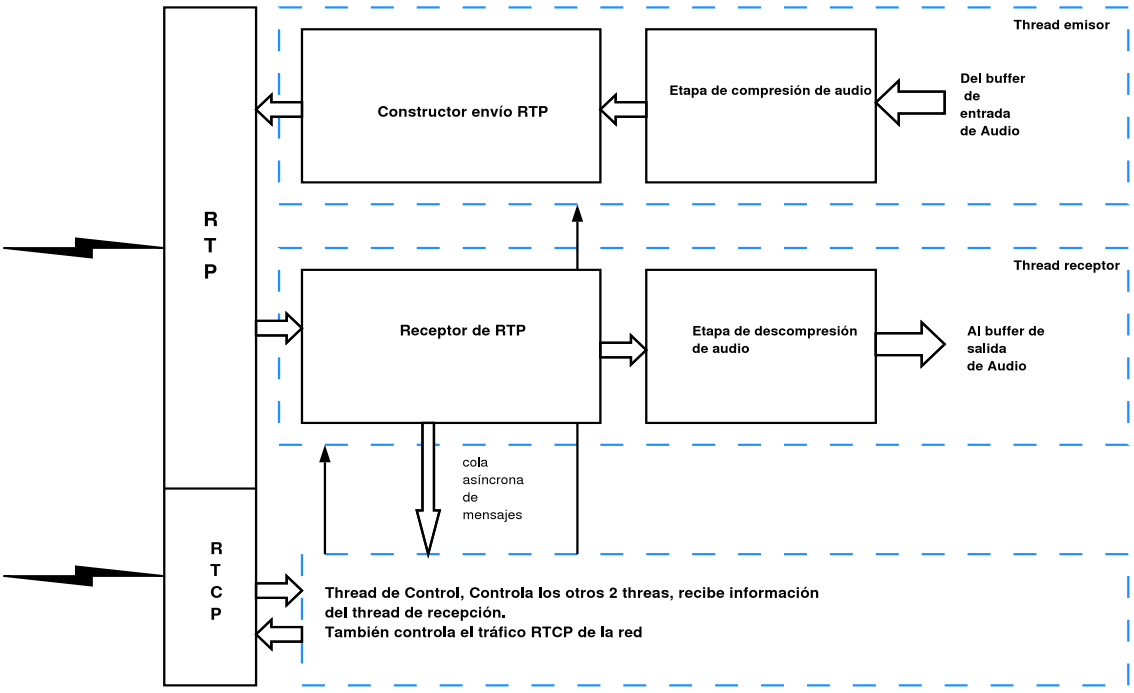

Figura 5.6: Funcionamiento del subsistema IO

de esta forma:

1. El usuario espera las llamadas en un puerto.
2. Cuando recibe una llamada comprueba que es para él, en caso contrario se descarta.
3. Si el usuario que llama está ignorado en la agenda de contactos, se rechaza la llamada
4. El usuario debe aceptar o rechazar la llamada, si la rechaza, finaliza la comunicación.
5. Una vez que se ha aceptado la llamada se negocian los codec que el usuario, que ha iniciado la llamada, ha elegido, si ambos usuarios no tienen codecs coincidentes, finaliza la comunicación.
6. Tras la negociación de los codecs, se procede a la negociación de los puertos de transmisión y recepción RTP.
7. Finalización de la negociación con éxito.

El diagrama 5.7 ilustra esta negociación:
A continuación se va a detallar todos los tipos de paquetes de SSIP:

***CONNECT_HELO*:** es el primer paquete que se envia para iniciar la negociación. Contiene información del usuario del que procede y al que va enviado. Soporta el envio de un mensaje de texto de no más de 256 caracteres.

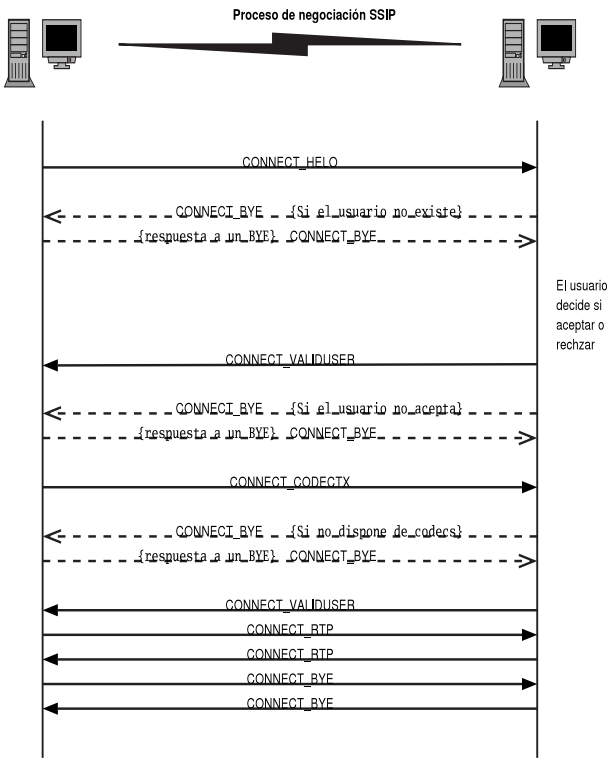

Figura 5.7: Negociación SSIP

***CONNECT_VALIDUSER*:** Indica que el usuario existe y ha aceptado la llamada. Si se recibe en este punto un *CONNECT_BYE* puede significar que el usuario no acepte la llamada o no exista en esa *maquina:puerto*

***CONNECT_CODECTX*:** Información de los codes de cada usuario, ordenados por orden de prioridad, se prueba sucesivamente con los codecs elegidos, hasta que se encuentre un codec que tienen ambos. cuando el segundo cliente encuentra una coincidencia, envía otro paquete *CONNECT_CODECTX* de vuelta sólo con la información de ese codec.

***CONNECT_RTP*:** Intercambio de información para los parámetros de la sesión RTP (puertos de TX y RX, etc). El segundo cliente debe responder con otro paquete de ese tipo.

***CONNECT_BYE*:** Finalización correcta de la negociación

Este protocolo presenta unas particularidades:

- Como se puede observar, cuando una máquina envía un paquete de tipo *CONNECT_BYE* la otra siempre debe responder, a modo de ping.

- La comunicación puede ser interrumpida en cualquier momento con el envío por cualquiera de las dos partes de un paquete *CONNECT_ERROR*. Este paquete tiene en le campo DATA, una descripción en formato texto del error ocurrido.

Los detalles de los campos y de la cabecera de este protocolo se muestran en el capítulo de *Implementación*.

5.4. Proxy SSIP

Este proceso es el encargado de gestionar el inicio y negociación de sesión para un conjunto de usuarios, es decir, aquellos que usen el proxy. Este sistema consta de dos subsistemas: el subsitema de negociación de inicio de sesión y el subsistema de control de usuarios. Ambos subsistemas están totalmente diferenciados, por lo que son totalmente independientes, excepto en la parte de gestión de usuarios, a la que ambos deben acceder para poder realizar su función. La figura 5.8 muestra gráficamente lo comentado.

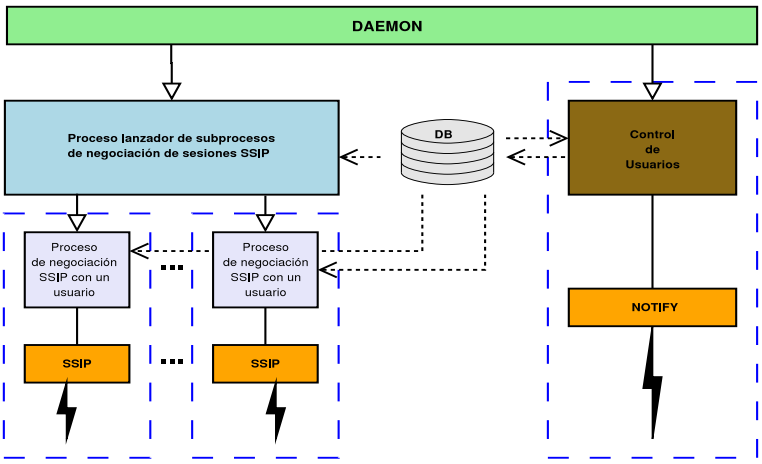

Figura 5.8: Funcionamiento global del proxy

La gestión de los usuarios activos se realiza almacenando sus datos en una base de datos. Como ambos subsistemas son independientes, se optó por un diseño concurrente con dos procesos, con un método de control de concurrencia, de otro modo puede ocurrir que, cuando un subsistema acceda al fichero para actualizar los datos de un usuario, el otro subsistema esté leyendo, provocando errores de coherencia en los datos, o incluso errores impredecibles. El diseño concurrente facilita que se pueda desabilitar cómodamente el subsistema de control de usuarios, manteniendo el otro subsistema funcionando, tal y como se especifica en los requisitos. También es necesario implementar un registro de las incidencias de todas las conexiónes a modo de *log*. El sistema de *logs*, tiene dos opciones

(de compilación): imprimir las incidencias por la salida estándar o volcarlas al sistema de logs que ofrece Linux (syslog).

5.4.1. Configuración

Todas las opciones que necesita el programa, las lee a partir del fichero de configuración, al arrancar. El fichero de configuración es el único parámetro que puede recibir el programa al ser lanzado y debe contener todas las siguientes opciones, todas ellas dentro de la sección *[SSIP_PROXY]*:

db_file : path absoluto del fichero que va a servir de base de datos de los usuarios del proxy. Si no existe será creado.

username_manage : nombre del proxy.

hostip_manage : interfaz de red (dirección IP) en la que esperará las peticiones de negociación de sesión SSIP.

port_manage : puerto en que escuchará el proceso de negociación SSIP.

num_forks_manage : opción de seguridad, indica el número máximo (límite) de procesos hijo -para atender negociaciones SSIP- que pueden ser creados.

username_notify : nombre del usuario virtual de notificación del proxy.

hostip_notify : interfaz de red (dirección IP) en la que será lanzado el proceso de control de usuarios con el protocolo *NOTIFY*.

port_notify : puerto en que escuchará el proceso de contol de usuarios (notificación con el protocolo *NOTIFY*).

A continuación se muestra un ejemplo del fichero de configuración típico:

```
#
# SSIP Proxy Configuration file
#
#

[SSIP_PROXY]

db_file=/home/riguera/PROYECTO/ssip_server/users_db.txt

username_manage=ssips
hostip_manage=192.168.1.2
port_manage=40000
num_forks_manage=10

username_notify=ssips
hostip_notify=192.168.1.2
port_notify=30000
```

Para procesar el fichero de configuración, se usa el mismo TDA definido para **Asubío**.

5.4.2. Subsistema de control de usuarios

También llamado subsistema de notificación. Su función es actualizar la base de datos del proxy, es decir, este subsistema controla los usuarios que emplean el proxy. Cada vez que un usario lanza su programa **Asubío**, éste notifica al proxy que el usuario ya está online y listo para recibir llamadas. En la notificación se envía el nombre del usuario, el nombre su máquina y el número de puerto SSIP (de ese usuario) en donde recibir las conexiones de negociación. Para ésta tarea se creó el protocolo *NOTIFY*.

La base de datos en donde se almacenan los usuarios del proxy, es el mismo TDA usado para la configuración de **Asubío**. Se optó por esa solución porque: cumple con los requisitos establecidos, se reutiliza el código y presenta una velocidad de proceso aceptable. El siguiente ejemplo muestra cómo es la base de datos por dentro:

```
[riguera]
host=192.168.1.2
port=10000
password=3jjlvwn
```

De esta forma, un administrador, puede editar fácilmente el fichero y borrar o inutilizar a determinados usuarios. Los resultados del ejemplo son interpretados de esta forma:

El usuario "riguera" se encuentra en el host "192.168.1.2" esperando negociaciones SSIP en el puerto 10000. El tercer campo es el "password" que aportó el usuario al iniciar su sesión con el proxy. Para que este usuario pueda borrar su registro en el proxy, deberá aportar el mismo "password". Esto es asi, para evitar que usuarios malintencionados borren a otros usuarios del proxy, lo cual provocaría que nunca podrían recibir llamadas.

Protocolo NOTIFY

Se desarrolló a partir de SSIP. Descripción de los campos de NOTIFY, de los paquetes que se envian al proxy:

***IPMODE*:** determina si el paquete contiene direcciones ip de IPv6 o IPv4. El valor de este campo modifica el tamaño de toda la estructura del paquete.

***FROM_USER*:** contiene el identificador de usuario que envía el paquete.

***FROM_ADDR*:** dirección IP del host en que se encuentra el usuario que ha enviado el paquete.

***FROM_PORT*:** número de puerto SSIP en el que se negociaran las sesiones para el usuario con el protocolo SSIP en su host.

***TO_USER*:** identificador al que va dirigido el paquete. Está reservado para futuro uso, para diferenciar distintos proxys. En esta versión el contenido de este campo es irrelevante.

***TO_ADDR*:** contiene la dirección IP del proxy al que va dirigido el paquete.

***TO_PORT*:** puerto del proxy al que va dirigido el paquete NOTIFY.

***DATA*:** contiene una contraseña generada aleatoriamente, que el proxy almacenará en su base de datos.

***TYPE*:** identifica el tipo de paquete.

El contenido de todos los campos *FROM_** y *TO_** de los paquetes NOTIFY que se reciben del proxy es el mismo que se envia -si todo transcurre correctamente-, pero permutado, es decir, los campos TO_* contendrán el valor que se había enviado antes en los campos FROM_* y viceversa.

En esta parte no es necesario indicar el tamaño de los campos, ya que es el diseño lógico. Además el tamaño varía, dependiendo de si los campos de las direcciones IP contienen IPv6 o IPv4. En el capítulo () se mostrará como se solucionó el problema de conflitos entre IPv4 e IPv6.

NOTIFY, dispone de los siguientes tipos de paquetes (campo TYPE), dependiendo del tipo de paquete, la información del resto de los campos puede cambiar de significado:

***USER_HELO_ONLINE*:** es el primer paquete que envía **Asubío**al proxy. Su misión es indicar que el usuario ya está online y listo para recibir llamadas.

***USER_OK_ONLINE*:** es la respuesta que envia el proxy, cuando se le ha enviado un *USER_HELO_ONLINE*, para indicar que el proceso ha ido bien y ese usuario ha sido añadido a la base de datos.

***USER_BYE_ONLINE*:** se envía para indicar que el usuario se da de baja en el proxy. Su misión es indicar que el usuario no está online y no va a admitir llamadas.

***USER_OK_OFFLINE*:** es la respuesta que envía el proxy tras un USER_BYE_ONLINE para indicar que el proceso ha ido bien y ese usuario ha sido dado de baja.

***USER_INCORRECT*:** es la respuesta que envia el proxy tras un USER_HELO_ONLINE o un USER_BYE_ONLINE. Si el proxy lo envía tras recibir un USER_HELO_ONLINE, significa que ya existe un usuario en la base de datos con el mismo nombre. Si es enviado tras un USER_BYE_ONLINE, se puede dar una de estas dos situaciones:

1. Que el usuario no exista en la base de datos.
2. Que la contraseña para ese usuario, indicada en el campo DATA, no sea la misma con la que inició la sesion, en este caso, el usuario no será borrado de la base de datos, continuando online.

***ERROR_PKT*:** este paquete lo envía el proxy cuando ocurre un error grave e inesperado (no encuentra la base de datos, no hay espacio en disco, etc.) y no puede llevar a cabo la operación solicitada por el paquete que ha recibido de **Asubío**. En este caso, en el campo DATA, se envía un *string* con la descripción del problema o error que motivó el fallo.

El contenido de todos los campos FROM_* y TO_* de los paquetes NOTIFY que se reciben del proxy es el mismo que se ha sido enviado, pero permutado, es decir, los campos TO_* contendrán ahora el valor que se había enviado antes en los campos FROM_* y

viceversa. NOTIFY funciona de forma simple, es un protocolo que sólo contiene dos transacciones: el cliente **Asubío**envia un paquete y el proxy siempre debe responder con otro paquete y fin de la transacción. El cliente siempre es el encargado de iniciar la conexión y el proxy de terminarla.

Funcionamiento

Debido que este subsistema no va a presentar mucha sobrecarga -un usuario sólo se notifica al darse de alta en el proxy o al abandonarlo- y las transaciones son muy cortas -sólo dos paquetes NOTIFY-, se ha diseñado para que atienda las notificaciones secuencialmente, es decir, hasta que no haya finalizado una notificación, no atiende a otra, lo que implica que no hay procesos concurrentes dentro del subsistema. Hay que considerar la posibilidad de que se quede *colgada* (a medias) una transacción, como este subsistema es secuencial, si esto ocurre, se bloquearía todo el proceso de control de usuarios.

El funcionamiento de este subsistema transcurre de este modo:

1. El proceso recibe un paquete NOTIFY procedente de un cliente. Si no es de tipo USER_HELO_ONLINE o USER_BYE_ONLINE o no reconoce el formato, se ignora y finaliza.

2. Si el paquete NOTIFY es de tipo USER_HELO_ONLINE, se busca el usuario en la base de datos. Si el usuario ya existe se constesta con un paquete USER_INCORRECT, en otro caso se almacena sus datos (usuario, host, puerto y clave) en la base de datos y se contesta con un paquete de tipo USER_OK_ONLINE.

3. Si el paquete NOTIFY es de tipo USER_BYE_ONLINE, se busca el usuario en la base de datos. Si el usuario existe y la clave del paquete coincide con la almacenada en la BD, se constesta con un paquete USER_OK_OFFLINE y se borran todos los datos de la BD. En otro caso se constesta con un paquete de tipo USER_INCORRECT y la BD no se modifica.

4. Si durante el proceso de notificación se produce algún error en el proxy, se envía al cliente un paquete de tipo ERROR_PKT, con el mensaje de error en el campo DATA.

5.4.3. Subsistema de negociación

Este subsitema se encarga de la negociación SSIP del inicio de sesión. Es, realmente, la parte que actúa de proxy, ya que canaliza la comunicación entre el programa cliente (del

usuario que llama) y el programa servidor (programa del usario al que se dirige la llamada -con el que se desea contactar-). El proceso de negociación es totalmente transparente para el usuario que inicia la llamada (figura 5.9).

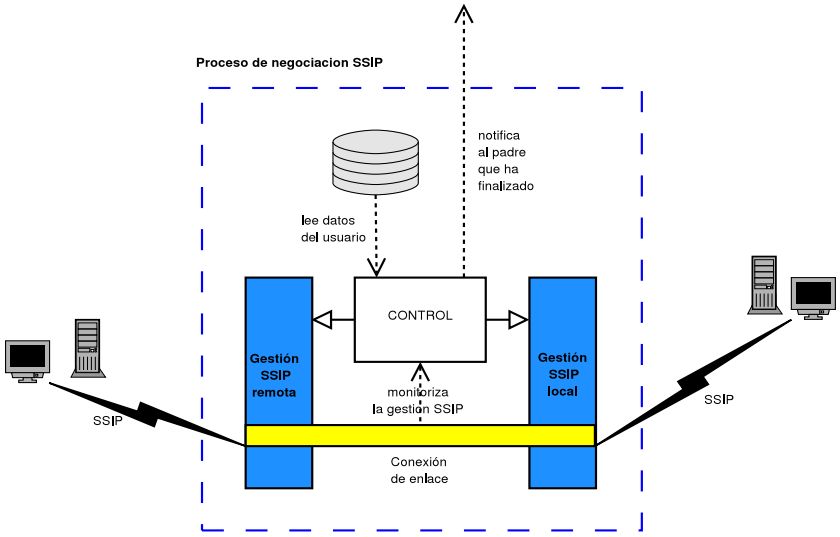

Figura 5.9: Negociacion SSIP a través del proxy

El proceso de negociación con proxy es compatible con la negociación sin proxy, es decir sigue el mismo diagrama de estados mostrado anteriormente en la figura (). Sucede de la siguiente forma:

1. El cliente envía un paquete SSIP de tipo CONNECT_HELO, que contiene información del usuario que realiza la llamada. En el paquete se indica el usuario con el que contactar y el host en donde está, que en este caso es el proxy.

2. El proxy recibe el paquete, comprueba que es de tipo CONNECT_HELO, y busca el usuario en su base de datos, si no lo encuentra, le reenvía al cliente un CONNECT_BYE y finaliza la comunicación.

3. Si el usuario existe en ese ámbito (está en la base de datos), el proxy obtiene los datos necesarios del usuario al que se dirige la llamada, es decir, obtiene el puerto y el host real. Crea una conexión con el programa del cliente llamado, que actuará como servidor, y le envía el paquete SSIP con el que se inicio todo este proceso.

4. Si el servidor (programa del cliente al que se llama) responde con un paquete CONNECT_VALIDUSER, el proxy crea un canal de comunicación bidireccional entre ambos programas. A partir de este momento el proxy, sólo se limita a actuar de enlace hasta que finalize la comunicación, suceda un error u ocurra un timeout. En

el caso de que ocurra un "timeout", antes de cortar la comunicación, se envía un CONNECT_ERROR al otro canal de enlace.

Como se puede apreciar, el proxy enmascara a todos los usuarios que contiene en la base de datos. Otra consideración que se ha tenido en cuenta es la carga de este subsistema. Al contrario que ocurre con el subsistema de negociación, atiende las peticiones secuencialemente -sin crear otros procesos hijos-, este subsistema crea un proceso hijo para cada petición que atiende.

5.4.4. Control de concurrencia

En este sistema no hay *threads*, sólo hay -como mínimo- dos procesos concurrentes que acceden a la base de datos. Para el control de concurrencia entre ambos procesos, en el acceso al fichero de la base de datos, el demonio *Proxy SSIP* crea un fichero ".lck" -vacío- que se bloquea para lectura o escritura por cada proceso concurrente, el tipo de bloqueo de este fichero, determina la operación que está teniendo lugar con el fichero de la base de datos de los usuarios.

5.4.5. Seguridad

Todo el **Proxy SSIP**, está diseñado teniendo en mente consideraciones de seguridad, por esa razón se realizan continuamente controles de datos y longitud de campos de los paquetes recibidos, "logeando" cualquier operación sospechosa mediante *syslog*. Pese a que este proceso no necesita privilegios especiales en el sistema -por esa razón se recomienda ejecutarlo como un usuario normal, no privilegiado- el lugar que ocupa en la red -en una pasarela o gateway- lo hace un blanco fácil para distintos tipos de ataques (overflow, DoS, etc.). También dispone de un control de número procesos hijo, para evitar ataques de *DoS* (Denegación de Servicio), de forma que sólo permite que existan como maximo el número de hijos que se indique en el fichero de configuración.

Capítulo 6

Implementación

Índice General

Como ya se comentó anteriormente, la implementación del sistema se realizó en C, porque es un lenguaje que genera ejecutables relativamente rápidos, lo cual es especialmente interesante en dominios como éste, donde hay que manejar datos en tiempo real.

Además, el sistema en el que se implementó el diseño ha sido *Linux*, pero, con las herramientas y bibliotecas utilizadas, ambos programas deberían compilar, casi sin cambios, en cualquier otro sistema que cumpla el estándar *POSIX*, incluso en sistemas *MS Windows* contruyendo un plugin para acceder al audio y cambiando el API de Sockets por el WinSock. Esto sería realmente difícil de conseguir realizando la implementación sobre un sistema cerrado, donde a veces resulta difícil saber si se están utilizando herramientas o librerías existentes sólo para ese sistema.

En el presente capítulo se mostrarán fragmentos de código fuente del software implementado, comentando los aspectos más significativos, de otro modo, resultaría imposible comentar en esta memoria las más de 22000 líneas de código fuente del software.

6.1. Organización en directorios

Todo el código fuente está organizado de tal forma que, cada directorio corresponde a un subsistema, excepto, todo el tratamiento de sonido que está en el directorio principal. El siguiente árbol muestra la estructura:

```
.
|-- codec_plugin
|   |-- adpcm
|   |   `-- adpcm
|   |-- gsm
|   |   `-- gsm-1.0-pl10
|   `-- sources
|-- conf
|-- crypt
|-- effect_plugin
|   `-- echo
|-- gui_gtk
|-- inout_plugin
|   `-- oss
|-- net
|-- pixmaps
```

```
|-- rtp
`-- ssip_proxy
```

Además esta estructura facilita la implementación de nuevos plugins, ya que para implementar uno nuevo, basta con crear un directorio dentro del tipo de plugin, modificar el Makefile para indicar el nuevo directorio y crear el código o algoritmo.

6.2. Algoritmos y arquitectura

No hubo que crear algoritmos complejos, ya que la mayoría del comportamiento queda definido por los RFC's. Sin embargo, hay ciertos subsistemas que implementan algoritmos simples pero interesantes, tales como: *buffer de audio*, *procesador de audio*, etc. que serán analizados posteriormente en la sección correspondiente.

Lo más llamativo de este proyecto es su arquitectura, con un alto grado de concurrencia -gracias a los *threads* o procesos ligeros-. El programa en reposo, sin ninguna sesión abierta, dispone de dos threads concurrentes: uno que controla los eventos de la GUI, interfaz gráfica, (el bucle *gtk_main()*) y otro que espera las posibles llamadas en un puerto SSIP. Cada vez que se crea una sesión con un usuario remoto, se generan tres nuevos threads que gestionan todo el proceso de envío, recepción y control del subsistema RTP/RTCP. Además el subsistema de procesamiento de audio, se ejecuta cuando hay al menos una sesión activa, para proveer de frames de audio a la/s sesión/es. También se crea otro thread temporal, cada vez que se realiza una llamada, que se encarga de gestionar toda la negociación SSIP, una vez que el proceso de llamada finaliza, el thread desaparece. En general, una fórmula para calcular el número de threads en un instante es:

$$Threads = \begin{cases} 2 & \text{si } N_s = 0 \\ 6 & \text{si } N_s = 1 \\ (N_s \cdot 3) + 3 & \text{en otro caso} \end{cases}$$

donde N_s es el número de sesiones activas. Si además, en ese instante, se está negociando una llamada desde el programa, habrá que sumar 1 al número de threads.

Esto en cuanto a la arquitectura de **Asubío**. La arquitectura del **Proxy SSIP**, también presenta un alto grado de concurrencia, pero en este caso, en forma de procesos, no threads. Se implementó de ese modo, porque, a pesar del mayor coste computacional de crear un proceso en lugar de un thread, se necesita independencia y seguridad. Cada vez que se atiende una negociación SSIP en el proxy, crea un proceso hijo que es totalmente independiente, si al hijo le ocurre algo, lo más grave que puede ocurrir es que se quede

el proceso *zombie* en el sistema. Además un administrador puede *matar* procesos que se queden colgados o que lleven varios minutos negociando la sesión, etc., es decir, permite un mayor control del proxy por parte del administrador.

6.3. Código compartido

En esta sección se analizará la implementación del código compartido tanto por **Asubío** como por **Proxy SSIP**.

6.3.1. Fichero de configuración

El fichero *configfile.h* define este TDA como:

```
struct _ConfigLine
{
    gchar *key;
    gchar *value;
};
typedef struct _ConfigLine ConfigLine;

struct _ConfigSection
{
    gchar *name;
    GList *lines;
};
typedef struct _ConfigSection ConfigSection;

struct _ConfigFile
{
    GList *sections;
    GMutex *write_mutex;
};
typedef struct _ConfigFile ConfigFile;

ConfigFile *cfgf_new (void);

ConfigFile *cfgf_open_file (gchar *filename, GError **error);
gboolean
cfgf_write_file (ConfigFile * cfg, gchar * filename, GError **error);

void cfgf_free (ConfigFile * cfg);

gboolean
cfgf_read_int (ConfigFile *cfg, gchar *section, gchar *key, gint *value);
gboolean
cfgf_read_string (ConfigFile *cfg, gchar *section, gchar *key, gchar **value);
gboolean
cfgf_read_float (ConfigFile *cfg, gchar *section, gchar *key, gfloat *value);
```

```
void
cfgf_write_int (ConfigFile *cfg, gchar *section, gchar *key, gint value);
void
cfgf_write_string (ConfigFile *cfg, gchar *section, gchar *key, gchar *value);
void
cfgf_write_float (ConfigFile *cfg, gchar *section, gchar *key, gfloat value);

gboolean cfgf_remove_key (ConfigFile *cfg, gchar *section, gchar *key);
```

Como se puede observar, el TDA *configfile* dispone de todas las funciones para acceder transparentemente al fichero de configuración del programa **Asubío**. Este mismo TDA es el que se usa en **Proxy SSIP** como fichero de configuración y también se usa para almacenar los usuarios activos, es decir de BD.

El semáforo *write_mutex* -línea 19- es el que permite que toda la estructura del TDA, pueda ser accedida concurrentemente por dos threads. En el siguiente listado se puede apreciar tal como se solucionó esa situación y como se usan las listas enlazadas -cada lista representa una sección- para almacenar y acceder a todos los datos.

```

gboolean
cfgf_read_string (ConfigFile *cfg, gchar *section, gchar *key, gchar **value)
{
    ConfigSection *sect;
    ConfigLine *line;

    g_mutex_lock(cfg->write_mutex);
    if (!(sect = _find_section(cfg, section))) {
      g_mutex_unlock(cfg->write_mutex);
      return FALSE;
    }
    if (!(line = _find_string(sect, key))) {
      g_mutex_unlock(cfg->write_mutex);
      return FALSE;
    }
    *value = g_strdup(line->value);
    g_mutex_unlock(cfg->write_mutex);
    return TRUE;
}
```

6.3.2. TDA's de acceso a red

Este TDA, se parece a un objeto, por esa razón en el capítulo de diseño se representó como tal. Ahora se puede apreciar como está definido un *Socket_tcp*. La estructura *Socket_udp* es idéntica, sólo se diferencia en las funciones que implementan. A continuación se muestra parte del fichero *tcp.h*, ya que este TDA es más completo que su correspondiente en UDP.

```
struct _Socket_tcp
{
    gint            mode;   /* IPv4 or IPv6 */
    gchar           *addr;
    guint16         rx_port;
    guint16         tx_port;
    ttl_t           ttl;
    fd_t            fd;
    struct in_addr  addr4;
#ifdef TCP_HAVE_IPv6
    struct in6_addr addr6;
#endif
};

typedef struct _Socket_tcp Socket_tcp;
```

Como se puede observar, el uso de comandos del prepocesador de C (compilación condicional) permite que todo el software se pueda compilar en máquinas que todavía no soporten IPv6. También se puede apreciar como se logró un nivel de abstracción por encima del API de *Sockets*, que permite el empleo de simúltaneo de IPv6 e IPv4. A continuación se muestra todo el API de *Socket_tcp*:

```
Socket_tcp
*tcp_init (const gchar *addr, const gchar *iface, guint16 rx_port,
           guint16 tx_port, gint ttl, GError **error);
void tcp_exit (Socket_tcp *s, GError **error);
gboolean
tcp_connect (Socket_tcp *s, const gchar *addr, guint16 port, GError **error);
Socket_tcp *tcp_accept (Socket_tcp *s, GError **error);

gint tcp_send (Socket_tcp *s, gpointer buffer, gint buflen, gint flags);
inline gint tcp_recv (Socket_tcp *s, gpointer buffer, gint buflen, gint flags);

gchar *tcp_host_addr (Socket_tcp *s, GError **error);

gint tcp_fd (Socket_tcp *s);
guint16 tcp_txport (Socket_tcp *s);
guint16 tcp_rxport (Socket_tcp *s);
gchar *tcp_addr (Socket_tcp *s);
```

Seguidamente se muestra una función típida del API de sockets, pero implementada con la abstracción entre IPv6 e IPv4:

```
Socket_tcp *tcp_accept (Socket_tcp *s, GError **error)
{
    struct sockaddr_in con4;
    gint fd;
    gint addrlen;
```

```
    gchar buf[100];
    Socket_tcp *st;
#ifdef TCP_HAVE_IPv6
    struct sockaddr_in6 con6;
#endif

    if (s == NULL) {
      g_set_error(error, TCP_ERROR, TCP_ERROR_NULL, "NULL");
      g_printerr("[TCP] tcp_accept: Socket_tcp = NULL\n");
      return NULL;
   }
    switch(s->mode) {
    case IPv4 :
        addrlen = sizeof(struct sockaddr_in);
      if (-1 == (fd = accept(s->fd, (struct sockaddr *) &con4, &addrlen))) {
          g_set_error(error, TCP_ERROR, TCP_ERROR_ERRNO, "%s", g_strerror(errno));
          g_printerr("[TCP] tcp_accept: %s\n", (*error)->message);
          return NULL;
      }
      st = (Socket_tcp *) g_try_malloc(sizeof(Socket_tcp));
      g_return_val_if_fail(st != NULL, NULL);
      st->mode = IPv4;
      memset(buf, 0, sizeof(gchar) * 100);
        inet_ntop(AF_INET, &con4.sin_addr, buf, sizeof(gchar) * 100);
      st->addr = g_strdup(buf);
      st->rx_port = s->rx_port;
      st->tx_port = con4.sin_port;
      st->ttl = s->ttl;
      st->fd = fd;
   break;
#ifdef TCP_HAVE_IPv6
    case IPv6 :
        addrlen = sizeof(struct sockaddr_in6);
      if (-1 == (fd = accept(s->fd, (struct sockaddr *) &con6, &addrlen))) {
          g_set_error(error, TCP_ERROR, TCP_ERROR_ERRNO, "%s", g_strerror(errno));
          g_printerr("[TCP] tcp_accept: %s\n", (*error)->message);
          return NULL;
      }
      st = (Socket_tcp *) g_try_malloc(sizeof(Socket_tcp));
      g_return_val_if_fail(st != NULL, NULL);
      st->mode = IPv6;
      memset(buf, 0, sizeof(gchar) * 100);
        inet_ntop(AF_INET6, &con6.sin6_addr, buf, sizeof(gchar) * 100);
      st->addr = g_strdup(buf);
      st->rx_port = con6.sin6_port;
      st->tx_port = s->tx_port;
      st->ttl = s->ttl;
      st->fd = fd;
      break;
#endif
    default :
      g_set_error(error, TCP_ERROR, TCP_ERROR_NOTIMPLEMENTED, "NULL");
      g_printerr("[TCP] tcp_accept: Socket type must be IPv4 or IPv6\n");
```

```
60        return NULL;
      }
      return st;
}
```

No todas las funciones fueron implementadas de esta forma con *switch/case*, la gran mayoría, tienen dos "sub-funciones", una para IPv4 y otra para IPv6, pareciéndose más a la metodología OO.

Como se puede apreciar, añadir el soporte para IPv6 ha sido trivial, ya que, en la mayoría de los casos basta con cambiar el parámetro *AF_INET* por *AF_INET6* en las funciones de sockets. El soporte para redes multicast se implementó de esta forma:

```
    if (IN6_IS_ADDR_MULTICAST(&(s->addr6)))
      {
      guint loop = 1;
5     struct ipv6_mreq  imr;
      imr.ipv6mr_multiaddr = s->addr6;
      imr.ipv6mr_interface = 0;

   if (setsockopt(s->fd, IPPROTO_IPV6, IPV6_ADD_MEMBERSHIP,
10       (char *) &imr, sizeof(struct ipv6_mreq)) != 0) {
           g_set_error(error, TCP_ERROR, TCP_ERROR_ERRNO, "%s", g_strerror(errno));
           g_printerr("[TCP] tcp_init6: %s\n", (*error)->message);
           g_free(s);
           return NULL;
15 }
   if (setsockopt(s->fd, IPPROTO_IPV6, IPV6_MULTICAST_LOOP,
       (char *) &loop, sizeof(loop)) != 0)  {
           g_set_error(error, TCP_ERROR, TCP_ERROR_ERRNO, "%s", g_strerror(errno));
           g_printerr("[TCP] tcp_init6: %s\n", (*error)->message);
20         g_free(s);
           return NULL;
   }
   if (setsockopt(s->fd, IPPROTO_IPV6, IPV6_MULTICAST_HOPS,
       (char *) &ttl, sizeof(ttl)) != 0) {
25         g_set_error(error, TCP_ERROR, TCP_ERROR_ERRNO, "%s", g_strerror(errno));
           g_printerr("[TCP] tcp_init6: %s\n", (*error)->message);
           g_free(s);
           return NULL;
   }
```

En el caso de IPv4, el procedimiento es similar.

6.3.3. Negociación *SSIP* y *NOTIFY*

En este apartado se comentarán algunos aspectos en la implementación de los protocolos SSIP y NOTIFY. Hay que tener en cuenta que las verdaderas intenciones eran usar

el protocolo SIP en lugar de SSIP (Simple SIP). Por otro lado, la implementación de estos protocolos no supuso ningún problema, basta con seguir los requisitos especificados en el capítulo de diseño.

```
/* ...  ssip.h */

#define SSIP_MANAGE_DEFAULT_PORT  10000
#define SSIP_NOTIFY_DEFAULT_PORT  20000
#define PKT_USERNAME_LEN     16
#define PKT_USERDES_LEN     256
#define PKT_USERDATA_LEN    512
#define PKT_HOSTNAME_LEN    256

// SSIP  (Simple SIP Protocol - protocolo simple de inicio de session)

struct _SsipPkt
{
    gchar to_user[PKT_USERNAME_LEN];
    gchar to_host[PKT_HOSTNAME_LEN];
    guint16 to_port;
    gchar from_user[PKT_USERNAME_LEN];
    gchar from_host[PKT_HOSTNAME_LEN];
    guint16 from_port;
    gint32 type;
    gint32 data_len;
    gchar data[PKT_USERDATA_LEN];
};
typedef struct _SsipPkt SsipPkt;

/* ... */

// SNP (Simple Notify Protocol - protocolo de notificacion simple) alias NOTIFY

struct _NtfPkt
{
    gint ipmode;
    gchar from_user[PKT_USERNAME_LEN];
    struct in_addr from_addr4;
#ifdef NET_HAVE_IPv6
    struct in6_addr from_addr6;
#endif
    guint16 from_port;
    gchar to_user[PKT_USERNAME_LEN];
    struct in_addr to_addr4;
#ifdef NET_HAVE_IPv6
    struct in6_addr to_addr6;
#endif
    guint16 to_port;
    gchar data[PKT_USERDES_LEN];
    gint32 type;
};
typedef struct _NtfPkt NtfPkt;
```

Sin embargo, lo más importante a la hora de diseñar un protocolo -después de lograr que cumpla su cometido- es su robusted. Ese punto lo cumplen ambos protocolos, se logró implementarlos de forma simple y robusta ya que admiten ser usados conjuntamente en redes IPv4 e IPv6. La robusted también viene dada en la implementación. Una implementación deficiente echaría al trate con un buen diseño. A continuación se muestran algunas funciones que añaden seguridad a la implementación:

```
/* ... ssip_neg.c */

static gboolean time_expired_mtime (Socket_tcp *skt, guint mt)
{
    fd_set rfds;
    struct timeval tv;
    gint valret, fd;

    fd = tcp_fd(skt);
    FD_ZERO(&rfds);
    FD_SET(fd, &rfds);
    tv.tv_sec = 0;
    tv.tv_usec = mt;
    valret = select(fd+1, &rfds, NULL, NULL, &tv);
    if (valret == -1) return FALSE;
    if (FD_ISSET(fd, &rfds)) return TRUE;
    return FALSE;
}
```

En el listado anterior, se muestra el uso de *select* para lograr que, en la lectura de datos de un socket, un proceso no se quede bloqueado indefinidamente. Esta función retorna *TRUE* si hay datos listos para leer en el socket y *FALSE* si ha transcurrido el tiempo especificado en μs y no se han recibido datos. Esta función también se emplea para que el usuario pueda interrumpir una negociación cuando desee: se introduce en un bucle con una variable que controle cuando el usuario desea salir.

En el siguiente listado se aprecian las comprobaciones de seguridad que se realizan cada vez que se recibe un paquete, para evitar desbordamientos de buffer, incoherencias en el tamaño de los campos, etc.

```
/* ... ssip_neg.c */

len = strlen(from_user) + 1;
if (PKT_USERNAME_LEN <= len) len = PKT_USERNAME_LEN;
g_strlcpy(pkt->from_user, from_user, len);
len = strlen(from_host) + 1;
if (PKT_HOSTNAME_LEN <= len) len = PKT_HOSTNAME_LEN;
g_strlcpy(pkt->from_host, from_host, len);
pkt->from_port = g_htons(from_port);

len = strlen(to_user) + 1;
```

```
if (PKT_USERNAME_LEN <= len) len = PKT_USERNAME_LEN;
g_strlcpy(pkt->to_user, to_user, len);
len = strlen(to_host) + 1;
if (PKT_HOSTNAME_LEN <= len) len = PKT_HOSTNAME_LEN;
g_strlcpy(pkt->to_host, to_host, len);
pkt->to_port = g_htons(to_port);

/* ... */
```

6.4. Plugins

En esta subsección se va a comentar las características de las funciones que deben tener implementadas los plugins para que funcionen correctamente.

6.4.1. *InOut* Plugin

El módulo debe tener implementada una función de tipo *LFunctionPlugin_InOut* con nombre *PLUGIN_INOUT_STRUCT* (definida en la línea 40) y devolver un puntero a una estructura *Plugin_InOut* para poder ser cargado y considerado como tal.

```
#define PLUGIN_INOUT_STRUCT "get_Plugin_InOut_struct"

struct _Plugin_InOut
{
    void *handle;
    gchar *name;
    gchar *filename;
    gchar *description;
    gchar *license;

    gint *is_usable;
    gint *is_selected;

    gint (*configure) (void);
    gint (*about) (void);
    gint (*init) (ConfigFile *cfgfile, GError **error);
    gint (*cleanup) (void);

    gint (*open_in) (gint open_flags, AFormat format, gint channels,
                     gint rate, gint fragmet, AInfo *info, GError **error);
    gint (*open_out) (gint open_flags, AFormat format, gint channels,
                      gint rate, gint fragmet, AInfo *info, GError **error);
    gint (*close_out) (void);
    gint (*close_in) (void);

    gint (*read) (gpointer buffer, gint l);
    gint (*write) (gpointer buffer, gint l);
```

```
30      gint (*get_in_volume) (gint *l_vol, gint *r_vol);
        gint (*set_in_volume) (gint l_vol, gint r_vol);
        gint (*get_out_volume) (gint *l_vol, gint *r_vol);
        gint (*set_out_volume) (gint l_vol, gint r_vol);

35      gint (*flush) (gint time);
        ACaps *(*get_caps) (void);
};
typedef struct _Plugin_InOut Plugin_InOut;

40 typedef Plugin_InOut *(*LFunctionPlugin_InOut) (void);
```

Esta interfaz define como son las funciones que debe tener la estructura que exportará el plugin cargado. No todas las funciones son oligatorias, a continuación se van a comentar sólo los símbolos que obligatoriamente debe exportar el módulo para que sea considerado válido:

***name**: nombre del plugin.

***filename**: nombre del fichero compilado.

***description**: breve descripción del plugin.

***license**: licencia bajo la cual se distribuye el plugin.

(*init): tras ser cargado el plugin, es la primera función que será llamada desde el programa principal. Recibe como parámetro un puntero a la configuración, retorna 1 si ha tenido exito, en caso contrario retorna un codigo negativo de error y rellena la estructura GError. Ésta es la primera función en ser llamada.

(*cleanup): esta función indica que el plugin puede liberar su memoria reservada, ya no va a ser usado.

(*open_in): función que inicializa el dispositivo de entrada de audio al programa con los parámetros especificados. Si es imposible establecer esos parámetros se devuelve un código negativo y se rellena la estructura GError con los motivos.

(*open_out): indéntica a la anterior, sólo que debe abrir el dispositivo de audio para salida. No existe un orden establecido para la llamada de *open_out* y *open_in*.

(*close_out): cierra el dispositivo de salida de audio, abierto con *open_out*.

(*close_in): cierra el dispositivo abierto con *open_in*.

(*read): lee el audio del dispositivo de entrada.

(*write): escribe el audio al dispositivo de salida.

(*get_caps): obtiene las capacidades del dispositivo de audio.

Como se puede apreciar, no es necesario que el dispositivo de audio disponga de controles de volumen (mixer). Si el dispositivo soporta full-duplex (se comprueba con *(*get_caps)* las llamadas de lectura y escritura de audio pueden ser concurrentes. También hay que destacar que este plugin almacena su estado internamente, usando variables *static*, con lo que, a diferencia de otros plugins, no es necesario que el programa principal conozca algo acerca de sus estados internos. Esto provoca que sólo pueda existir sólo una "instancia" de este plugin en el sistema.

Para la implementación práctica de este plugin se usó el sistema OSS, ya que se usa mayoritariamente como módulo de acceso a la tarjeta de sonido en el kernel de Linux. Además soporta perfectamente full-duplex, condición necesaria para la mayoría de usuarios que sólo disponen de una tarjeta de sonido en su PC.

6.4.2. *Effect* Plugin

Definición de la interfaz que deben cumplir los plugins de effectos de sonido:

```
enum _TPCall
{
    AUDIO_CALL_NONE = 0,
    AUDIO_CALL_IN = 1 << 0,
    AUDIO_CALL_OUT = 1 << 1,
};
typedef enum _TPCall TPCall;

#define PLUGIN_EFFECT_STRUCT "get_Plugin_Effect_struct"

struct _Plugin_Effect
{
    void *handle;
    gchar *name;
    gchar *filename;
    gchar *description;
    gchar *license;

    gint *is_usable;
    gint *is_selected;

    gpointer (*init) (ConfigFile *cfgfile, AudioProperties *aprts, GError **error);
    gint (*cleanup) (gpointer status);

    gint (*configure) (gpointer status);
    gint (*about) (void);

    gint (*pefunction) (gpointer status, gpointer d, gint *length, TPCall type);
```

```
30 };
typedef struct _Plugin_Effect Plugin_Effect;

typedef Plugin_Effect *(*LFunctionPlugin_Effect) (void);
```

El plugin debe tener definida una función *PLUGIN_EFFECT_STRUCT* de tipo *LFunctionPlugin_Effect_InOut* que devolverá un puntero a una estructura *Plugin_Effect* para poder ser cargado. Al igual que en el plugin anterior, se detallarán sólo los símbolos mínimos que debe exportar.

***name**: nombre del plugin.

***filename**: nombre del fichero compilado.

***description**: breve descripción del plugin.

***license**: licencia bajo la cual se distribuye el plugin.

(*init): tras ser cargado el plugin, es la primera función que será llamada desde el programa principal. Recibe como parámetro un puntero a la configuración y las propiedades del audio que va recibir, retorna un puntero a *void* (una estructura desconocida, su estado interno) si ha tenido éxito, en caso contrario debe retornar un NULL y rellenar la estructura GError. Ésta es la primera función del plugin en ser llamada.

(*cleanup): esta función indica que el plugin puede liberar su memoria reservada, ya no va a ser usado.

(*pefunction): función de tratamiento del audio. El primer parámetro que recibe es su estado interno, el segundo es un puntero a un buffer de audio -que puede ser modificado-, el tercero, es el tamaño del buffer de audio y el último, de tipo *TPCall* indica de donde procede el audio del buffer, si es audio de entrada valdrá *AUDIO_CALL_IN* y para el audio de salida, *AUDIO_CALL_OUT*. La función debe tener capacidad de decidir qué hacer en cada caso.

Hay dos consideraciones a tener en cuenta:

1. los frames de audio que recibe la función no siempre tienen la misma separación temporal, es decir, pueden llegar varios seguidos, dependiendo de la congestión de la red, etc.

2. a este plugin van a estar accediendo concurrentemente dos threads -uno de tratamiento del audio de entrada (*TPCall* = *AUDIO_CALL_IN*) y otro de salida (*TPCall*

= *AUDIO_CALL_OUT*)-, cada uno almacenará un estado diferente, esto significa que no es recomendable usar variables *static* en *(*pefunction)*.

Como implementación práctica de este plugin se codificó una librería de eco: "libecho.so". Simplemente añade un efecto de eco al audio. El algoritmo de eco se basa en sumar el frame de audio actual con el anterior pero reduciendo el volumen del anterior, para que no degrade seriamente el actual. Jugando con el volumen de mezcla, con retardos y con otros frames anteriores se pueden lograr efectos impresionantes y de esos parámetros sale su configuración.

6.4.3. *Codec* Plugin e implementación OO de herencia

A continuación se muestra la definición del API para el plugin de compresión-descompresión de audio:

Listing 6.1: API del *Codec* Plugin

```
enum _CodecMode
{
    CODEC_NONE = 0,
    CODEC_ENCODE = 1 << 0,
    CODEC_DECODE = 1 << 1,
};
typedef enum _CodecMode CodecMode;

#define PLUGIN_CODEC_STRUCT "get_Plugin_Codec_struct"

struct _Plugin_Codec
{
    void *handle;
    gchar *name;
    gchar *filename;
    gchar *description;
    gchar *license;
    gint uncompressed_fr_size;
    gint compressed_fr_size;
    gint rate;
    gint bw;
    gint payload;
    gint audio_ms;
    gint *is_usable;
    gint *is_selected;
    struct _Codec *(*constructor) (void);
};
typedef struct _Plugin_Codec Plugin_Codec;

typedef Plugin_Codec *(*LFunctionPlugin_Codec) (void);
```

Es importante recordar, que el diseño de este plugin es diferente a los anteriores. En este caso, se ha implementado un sistema de OO con herencia. Cada plugin de este tipo es una *especialización* de la clase *Codec*, la clase de la que heredan estos métodos:

Listing 6.2: Métodos que se especializarán en las subclases

```
struct _Codec
{
    gboolean (*_init) (struct _Codec *codec, AudioProperties *ap,
             ConfigFile *config, CodecMode mode, GError **error);
    gboolean (*_getinfo) (struct _Codec *codec, Plugin_Codec *info);

    gint (*_encode) (struct _Codec *codec, gpointer frame, gpointer data);
    gint (*_decode) (struct _Codec *codec, gpointer data, gpointer frame);

    gint (*_about) (struct _Codec *codec);
    gint (*_configure) (struct _Codec *codec);
    gboolean (*_destroy) (struct _Codec *codec);
};
typedef struct _Codec Codec;

inline gboolean codec_init (Codec *codec, AudioProperties *ap,
                            ConfigFile *config, CodecMode mode, GError **error);
inline gboolean codec_getinfo (Codec *codec, Plugin_Codec *info);
inline gint codec_encode (Codec *codec, gpointer frame, gpointer data);
inline gint codec_decode (Codec *codec, gpointer data, gpointer frame);
inline gint codec_about (Codec *codec);
inline gint codec_configure (Codec *codec);
inline gboolean codec_destroy (Codec *codec);

inline gboolean codec_is_usable (Plugin_Codec *codec, double bandwidth);
```

Todos los métodos de la clase *Codec* se implementan de esta forma:

Listing 6.3: Implementación de los métodos del listado anterior

```
#include "codec.h"

inline gboolean
codec_init (Codec *codec, AudioProperties *ap,
            ConfigFile *config, CodecMode mode, GError **error)
{
    GError *tmp_error = NULL;
    gboolean val;

    val = codec->_init(codec, ap, config, mode, &tmp_error);
    if (tmp_error != NULL) {
      g_propagate_error(error, tmp_error);
      return FALSE;
    }
    return val;
```

```
}

inline gboolean codec_getinfo (Codec *codec, Plugin_Codec *info)
{
    return codec->_getinfo(codec, info);
}

inline gint codec_encode (Codec *codec, gpointer frame, gpointer data)
{
    return codec->_encode(codec, frame, data);
}

inline gint codec_decode (Codec *codec, gpointer data, gpointer frame)
{
    return codec->_decode(codec, data, frame);
}

inline gint codec_about (Codec *codec)
{
    return codec->_about(codec);
}

inline gint codec_configure (Codec *codec)
{
    return codec->_configure(codec);
}

inline gboolean codec_destroy (Codec *codec)
{
    return codec->_destroy(codec);
}
```

De forma que, a la hora de implementar un plugin en concreto, se procede de esta forma:

Listing 6.4: Fragmento de implementación del plugin *ADPCM*, libadpcm.so

```
struct _ADPCMCodec
{
    Codec codec_class;  /* codec heredado, siempre en primer lugar */
    CodecMode mode;     /* para luego poder hacer un casting       */
    Adpcm_State adpcm_encode_status;
    gint adpcm_encode_len;
    Adpcm_State adpcm_decode_status;
    gint adpcm_decode_len;
};
typedef struct _ADPCMCodec ADPCMCodec;

/* ... */

Codec *adpcm_codec_new ()  /* constructor de la clase ADPCMCodec */
{
```

```
    ADPCMCodec *obj;

    obj = (ADPCMCodec *) g_malloc(sizeof(ADPCMCodec));
    obj->codec_class._init = &adpcm_codec_init;
   //obj->codec_class._getinfo = &adpcm_codec_getinfo;
    obj->codec_class._getinfo = NULL;
    obj->codec_class._encode = NULL;
    obj->codec_class._decode = NULL;
    obj->codec_class._about = &adpcm_codec_about;
    obj->codec_class._configure = &adpcm_codec_configure;
    obj->codec_class._destroy = &adpcm_codec_destroy;
    obj->mode = 0;
    obj->adpcm_encode_len = 0;
    obj->adpcm_decode_len = 0;

    return ((Codec *) obj);
}

gboolean
adpcm_codec_init (Codec *codec, AudioProperties *ap,
                  ConfigFile *config, CodecMode mode, GError **error)
{
    ADPCMCodec *obj = (ADPCMCodec *) codec;

    UNUSED(config);
    if (ap->format != FMT_S16_LE) {
      g_set_error(error, PLUGIN_CODEC_AUDIO_ERROR,
                    PLUGIN_CODEC_AUDIO_ERROR_APROPERTIES,
                    "audio_con_formato_no_soportado");
        return FALSE;
    }
    if (mode & CODEC_DECODE) {
      if (mode & CODEC_ENCODE) {
          obj->codec_class._encode = &adpcm_codec_encode;
          obj->adpcm_encode_len = ADPCM_ENCODE_LENGTH;
      }
      obj->codec_class._decode = &adpcm_codec_decode;
      obj->adpcm_decode_len = ADPCM_DECODE_LENGTH;
      obj->mode |= mode;
       return TRUE;
    } else {
      if (mode & CODEC_ENCODE) {
          obj->codec_class._encode = &adpcm_codec_encode;
          obj->adpcm_encode_len = ADPCM_ENCODE_LENGTH;
      }
      obj->mode |= mode;
       return TRUE;
    }
    return FALSE;
}
```

Como se puede apreciar en este úlimo listado en la línea 19, el constructor del codec *ADPCM* crea un objeto de tipo *ADPCMCodec*, pero retorna realmente (línea 32) un

objecto de tipo *Codec*, haciendo un *casting*. Esto se puede hacer así, porque la estructura *ADPCMCodec* tiene incluída en primer lugar a la estructura *Codec*, con lo que el casting sólo reconocerá esa estructura. De esta forma se consigue una referencia a memoria en la superclase *Codec*, de tipo *codec*, pero que apunta realmente a una estructura *ADPCMCodec* que nunca va a ser accedida, ya que en ese nivel, no se conoce su formato ni existencia. Posteriormente, como se aprecia en los listados 6.2, 6.3 y 6.4, todas las llamadas de las clases heredadas reciben siempre una refencia a *Codec*, pero, sólo cada subclase especializada conoce realmente la estructura interna de *ADPCMCodec*.

Tras explicar el funcionamiento de la herencia, se va a detallar el uso de cada una de las funciones que debe tener implementadas un *Codec Plugin*. El API está mostrada en el listado 6.1 .

El plugin debe tener definida una función *PLUGIN_CODEC_STRUCT* de tipo *LFunctionPlugin_Effect_InOut* que devolverá un puntero a una estructura *Plugin_Codec* para poder ser cargado. Estos son los símbolos que debe exportar el plugin tras la llamada a la función definida por *PLUGIN_CODEC_STRUCT*:

***name**: nombre del plugin.

***filename**: nombre del fichero compilado.

***description**: breve descripción del plugin.

***license**: licencia bajo la cual se distribuye el plugin.

uncompressed_fr_size: tamaño del frame de audio que debe recibir el plugin, el programa principal por medio del doble buffer puede ofrecerle cualquier tamaño.

compressed_fr_size: tamaño del frame de audio comprimido.

rate: frecuencia a la que debe recibir el audio (Hz).

bw: ancho de banda aproximado que consume.

payload: número que indentifica el codec en el estándar RTP/RTCP (RFC 1889).

audio_ms: tiempo en milisegundos de cada frame de audio comprimido.

(*constructor): método constructor de la clase, debe devolver una referencia a una estructura *codec* (no NULL).

A continuación se detalla la estructura *codec* que implementa la superclase para la herencia a las subclases (plugins):

1. **(*_init)**: el programa principal, tras llamar al constructor, llama a *init* con los parámetros: *codec*, el codec construido; *ap*, las propiedades del audio y el modo de compresión elegido: *CODEC_NONE* si sólo se inicia el plugin para ser configurado, *CODEC_ENCODE* si sólo va a codificar audio, *CODEC_DECODE* si sólo se va a codificar audio y *CODEC_DECODE — CODEC_DECODE* para ambas cosas.

2. **(*_getinfo)**: obtiene toda la información acerca del codec: estado, caracteristicas, etc.

3. **(*_encode)**: función que se llama para comprimir un buffer de audio de tamaño *uncompressed_fr_size* a *compressed_fr_size*.

4. **(*_decode)**: se llama para descomprimir un frame de audio, "inversa" de la anterior.

5. **(*_destroy)**: destrucción del objecto.

El resto de las funciones no son de implementación obligada. Hay, también, que tener en cuenta que, a igual que ocurría en el plugin explicado anteriormente, las funciones de compresión y descompresión pueden ser llamadas concurrentemente y no siempre con los mismos intervalos temporales.

Para la implementación práctica de este plugin se usó el algoritmo de ADPCM, ya que su código es libre, relativamente corto y es relativamente fácil de entender. Logra comprimir el audio en trozos de 256 bytes consiguiendo una relación de compresión 4:1. No posee sistema de VAD (Voice Activity Detection) ni complejos sistemas de predicción de audio. El plugin se llama "libadpcm.so".

6.5. Procesador de Audio

Es un subsistema clave en **Asubío**, permite el desarrollo -fácilmente- del concepto de multisesión, ya que es el encargado de mezclar y/o proveer audio de/a varias fuentes/-consumidores. Es un hilo independiente que controla el plugin *InOut*. Recibe y despacha el audio por *colas asíncronas* bloqueantes en lectura, lo que permite una implementación fácil a modo FIFO. Internamente su funcionamiento se basa en mecanismos del kernel de IPC (gestión de memoria compartida y colas de mensajes entre procesos), concretamente en el paso de *mensajes*, pero consumiendo muchos menos recursos que su implementación

con FIFO's. Este subsistema abstrae el dispositivo de sonido para ofrecer mayores prestaciones al programa: cambio de tamaño de bloque de audio, cambio de formato interno de audio, mezclador/dispensador de audio, etc. A continuación se muestra el API que implementa este TDA.

```
struct _NodeAqueue
{
    gint id_aqueue;
    GAsyncQueue *aqueue;
};
typedef struct _NodeAqueue NodeAqueue;

gboolean
audio_processor_loop_create (AudioProperties *ap_dsp,
                             AudioProperties *ap_internal,
                             gint blocksize_dsp, gint blocksize_internal,
                             glong block_time, gint *cng,
                             Plugin_InOut *pluginio, GError **error);
gboolean audio_processor_loop_destroy (GError **error);

gboolean audio_processor_loop_create_mic (gint id, GAsyncQueue **queue);
gboolean audio_processor_loop_free_mic (gint id);

gboolean audio_processor_loop_create_spk (gint id, GAsyncQueue **queue);
gboolean audio_processor_loop_free_spk (gint id);

gpointer audio_processor_loop (gpointer data);
```

Seguidamente se muestra la implementación clave del TDA, el thread que hace posible su funcionamiento:

```
gpointer audio_processor_loop (gpointer data)
{
    guchar *areturn, *buffer_read, *buffer_read_aux, *buffer_write;
    guchar *buffer_write_add, *buffer_write_aux;
    NodeAqueue *node;
    gint number_spk, counter, bytes_read, c, counterpops;
    gfloat factor, tmp;
    gint16 *j, *k;
    gpointer buf;

    UNUSED(data);
    buffer_read = (guchar *) g_malloc0(audio_blocksize_dsp);
    buffer_read_aux = (guchar *) g_malloc0(audio_blocksize_buf);
    buffer_write_aux = (guchar *) g_malloc0(audio_blocksize_buf);
    buffer_write_add = (guchar *) g_malloc0(audio_blocksize_buf);
    buffer_write = (guchar *) g_malloc0(audio_blocksize_dsp);

    while (process) {
      if (sleep_time != 0) {
```

```
            sleep_us(sleep_time - 1000);
        }
         g_mutex_lock(mutex_aqueue_mic);
        bytes_read = plugin->read(buffer_read, audio_blocksize_dsp);
        if (bytes_read != audio_blocksize_dsp) {
            g_printerr("[AUDIO_PROCESSOR] Bloque esperado != bytes leidos
(%d != %d)\n", audio_blocksize_dsp, bytes_read);
            if (bytes_read <= 0) {
              g_printerr("[AUDIO_PROCESSOR] bytes leidos : %d\n", bytes_read);
            } else {
              gint cond = audio_blocksize_dsp - bytes_read;
              gint index = bytes_read;
              while (cond > 0) {
                  bytes_read = plugin->read(&buffer_read[index], cond);
                  index += bytes_read;
                  cond = cond - bytes_read;
              }
            }
        }
         memcpy(buffer_read_aux, buffer_read, audio_blocksize_dsp);
        buf = buffer_read_aux;
        if (audio_in_convert_function != NULL)
           audio_blocksize_buf = audio_in_convert_function(buf,audio_blocksize_dsp);
        else audio_blocksize_buf = audio_blocksize_dsp;

        // Audio in (mic)
        counter = 0;
        node = (NodeAqueue *) g_slist_nth_data(list_aqueues_mic, counter);
        while (node != NULL) {
#ifdef A_PROCESSOR_DEGUG
            g_print("[AUDIO_PROCESSOR] LEE: %d; Numero de cola: %d;
long cola mics: %d\n", bytes_read, counter,
                      g_async_queue_length(node->aqueue));
            fflush(NULL);
#endif
            g_async_queue_push(node->aqueue, g_memdup(buffer_read_aux,
                               audio_blocksize_buf));
            node = (NodeAqueue *) g_slist_nth_data(list_aqueues_mic, ++counter);
        }
        g_mutex_unlock(mutex_aqueue_mic);

        // Audio out (speaker)
        g_mutex_lock(mutex_aqueue_spk);
        counter = 0;
        counterpops = 0;
        number_spk = g_slist_length(list_aqueues_spk);
        factor = 1.0 / (gfloat) number_spk;
        memset(buffer_write_add, '\0', audio_blocksize_buf);
        node = (NodeAqueue *) g_slist_nth_data(list_aqueues_spk, counter);
        while (node != NULL) {
            if ((areturn = g_async_queue_try_pop(node->aqueue)) != NULL) {
              counterpops++;
              if (number_spk != 1) {
```

```
                memcpy(buffer_write_aux, areturn, audio_blocksize_buf);
                j = (gint16 *) buffer_write_aux;
                k = (gint16 *) buffer_write_add;
                for (counter=0; counter<(audio_blocksize_buf/2); counter++) {
#if G_BYTE_ORDER == G_LITTLE_ENDIAN
                  tmp = *j * factor;
                  if (tmp >= 0) *k = *k + (gint16) (tmp + 0.5);
                  else *k = *k + (gint16) (tmp - 0.5);
#elif G_BYTE_ORDER == G_BIG_ENDIAN
                  tmp = GINT16_TO_BE(*j);
                  tmp = tmp * factor;
                  if (tmp >= 0)
                     *k = GINT16_TO_LE(GINT16_TO_BE(*k) + (gint16) (tmp + 0.5));
                  else
                     *k = GINT16_TO_LE(GINT16_TO_BE(*k) + (gint16) (tmp - 0.5));
#else
#error "G_BYTE_ORDER should be big or little endian."
#endif
                  j++;
                  k++;
                }
            } else {
                memcpy(buffer_write_add, areturn, audio_blocksize_buf);
            }
            g_free(areturn);
          }
#ifdef A_PROCESSOR_DEGUG
          g_print("[AUDIO_PROCESSOR] ESCRIBE: %d; Numero de cola: %d;
 ____________________long cola speakers %d\n", bytes_read, counter,
                  g_async_queue_length(node->aqueue));
        fflush(NULL);
#endif
          node = (NodeAqueue *) g_slist_nth_data(list_aqueues_spk, ++counter);
      }
      buf = buffer_write_add;
      if (audio_out_convert_function != NULL)
         audio_blocksize_dsp =audio_out_convert_function(buf,audio_blocksize_buf);
      memcpy(buffer_write, buffer_write_add, audio_blocksize_dsp);
      if ((counterpops == 0) && (*audio_cng == 1)) {
          j = (gint16 *) buffer_write;
          for (c=0; c<(audio_blocksize_dsp/2); c++) {
            tmp = (*j * (1 - AP_GNG_FACTOR)) + (AP_GNG_FACTOR * (gint16)
            g_random_int_range(-AP_CNG_RANGE, AP_CNG_RANGE));
            if (tmp >= 0) *j = (gint16) (tmp + 0.5);
            else *j = (gint16) (tmp - 0.5);
            j++;
          }
      }
      bytes_read = plugin->write(buffer_write, audio_blocksize_dsp);
      if (bytes_read != audio_blocksize_dsp) {
          g_printerr("[AUDIO_PROCESSOR] Bloque disponible != bytes escritos
 ____________________(%d != %d)\n", audio_blocksize_dsp, bytes_read);
      }
```

```
        g_mutex_unlock(mutex_aqueue_spk);
        sleep_us(1000);
    }
    plugin->flush(0);
    g_free(buffer_read);
    g_free(buffer_read_aux);
    g_free(buffer_write_aux);
    g_free(buffer_write_add);
    g_free(buffer_write);
    g_thread_exit(NULL);
    return NULL;
}
```

Toda esta función constituye un thread que es lanzado ante cualquier operación de E/S de audio mediante el *InOut Plugin* seleccionado. Como se puede observar en las líneas 69 y 48 y en el listado del API, este subsistema controla las colas consumidoras y productoras de audio con listas enlazadas que, en el primer caso recorre para colocar un buffer de audio en cada una y en el segundo suma el audio de todas las fuentes de la lista. Ambas operaciones se realizan bloqueando sendos semáforos, evitan que mientras se esté recorriendo cada lista, otro proceso concurrente cree otra cola que se añadiría a la lista en cuestión, provocando errores incontrolables.

Por otro lado, hay que decir que, en las colas de asíncronas de audio, sólo se envían y reciben referencias a buffers, creadas dinámicamente. En un primer analísis, se podría pensar que esta implementación reduce el rendimiento, pero eso no es del todo cierto:

1. El número de reservas de memoria por segundo depende del tamaño de bloque de audio. Para un tamaño de bloque de 1024 bytes a 8000 Hz. se produce una reserva de memoria cada 0,128 segundos, un tiempo aceptable.

2. Como el número de reservas/liberaciones de memoria se mantiene constante -cada vez que se crea un nuevo buffer, se libera otro-, las llamadas de reserva de memoria no llegan al kernel, las controla *libc*, por lo que, el rendimiento no empeora significativamente.

3. Este sistema permite absorber ilimitadamente las variaciones temporales en el envío/-recepción de paquetes de audio -provocadas en la red-, ya que, cada bloque de audio, tiene un espacio de memoria diferente. Con un sistema de buffers estáticos, esto no sería posible -o por lo menos tan fácil de implementar- porque, ante un cierto retardo, se sobreescribirían buffers no procesados.

Otra consideración importante, es que, este thread es capaz de controlar el tiempo que debe esperar entre lecturas/escrituras en el dispositivo de audio. La mejor forma de medir

el tiempo de forma precisa es a través del propio dispositivo de lectura, ya que bloquea la llamada hasta que hay un bloque listo para leer. Sin embargo si el plugin de E/S no tiene lectura bloqueante, se calcula el tiempo que le corresponde y duerme el proceso (línea 20) simulando un bloqueo. Asimismo si el formato de sonido que ofrece el *InOut Plugin* no coincide con el tratado internamente (16 bits, mono, 8000 Hz), dispone de funciones que convierten la entrada (línea 42) y posteriormente la salida (línea 109) entre ambos formatos de audio. La función de conversión se obtiene al inicilizar todo el subsistema, y es un puntero a una función disponible en el fichero *audio_convert.h*.

También se puede observar que el código incorpora un sistema de CNG (Comfort Noise Generation) -línea 112- para producir un ligero ruido, que simule que la comunicación está activa.

6.6. Doble buffer de audio

Esta es otra parte fundamental en **Asubío**. Tiene como función principal, servir de adaptador entre el bloque de audio que aporta el subsistema *Procesador de Audio* y el bloque de audio que solicita un plugin. Normalmente el primero es mucho mayor que el segundo, por lo que hay que implementar un sistema que consiga gestionar eficientemente este proceso. El buffer lee el audio de una cola asícrona, que se ha solitado al subsistema *Procesador de Audio*- y envía ese audio al codec compresor, para más tarde despacharlo por la red en formato RTP, esto es todo lo que realiza un thread, pero paralelamente hay otro que realiza la función inversa.

Además, también realiza otras funciones como:

- Controlar el tamaño de las colas de audio que enlazan con el Procesador de Audio. Si este buffer deja de leer/escribir, las colas se vaciarán y se sincronizará inmediatamente el audio. Por esa razón el "mute" ofrece a su vez la posibilidad de sincronizar el audio. Si hay muchos bloques en la cola, se produce un retardo en el sonido, pero esto, por supuesto tiene las ventajas ya mencionadas anteriormente.
- Gestionar los plugins de efectos de audio. En estos buffers es donde se llama a las funciones de los *Effect Plugin*, para cada bloque de audio que se lee de la cola asíncrona de recepción/envío (línea 95).

```
/* ... audio_buffer.h */

struct _AudioBuffer
{
```

```
    guchar *buffer;
    gint ptr_write_buffer;
    gint ptr_read_buffer;
    gint blocksize_write;
    gint blocksize_read;
    gint buffer_length;
    AudioProperties *pp;
};
typedef struct _AudioBuffer AudioBuffer;

struct _PluginEffectNode
{
    Plugin_Effect *plugin;
    GModule *module;
    gint id;
    gboolean active;
    gpointer state;
};
typedef struct _PluginEffectNode PluginEffectNode;

struct _ABuffer
{
    GSList *list_plugins;
    GAsyncQueue *aqueue;
    AudioBuffer *audio;
};
typedef struct _ABuffer ABuffer;

ABuffer
*init_read_audio_buffer (gpointer audio_aqueue, gint audio_blksize_in,
                  gint audio_blksize_out, AudioProperties *apts,
                  GSList *plugins, GError **error);
gint free_read_audio_buffer (ABuffer *ab);
gint read_audio_buffer (ABuffer *ab, gint *mute, gpointer audio_data);

ABuffer
*init_write_audio_buffer (gpointer audio_aqueue, gint audio_blksize_in,
                  gint audio_blksize_out, AudioProperties *apts,
                  GSList *plugins, GError **error);
gint free_write_audio_buffer (ABuffer *ab);
gint write_audio_buffer (ABuffer *ab, gfloat *f, gint *mute,
                  gpointer audio_data);

/* --- */
/* ... audio_buffer_out.c */

gint
write_audio_buffer (ABuffer *ab, gfloat *f, gint *mute, gpointer audio_data)
{
    gpointer c;
    gint16 *j;
    gfloat tmp, factor;
    gint b, counter;
```

```
    PluginEffectNode *pn;

    factor = *f;
    memcpy((ab->audio->buffer + ab->audio->ptr_write_buffer), audio_data,
        ab->audio->blocksize_write);
    ab->audio->ptr_write_buffer += ab->audio->blocksize_write;
    if ((ab->audio->ptr_write_buffer - ab->audio->ptr_read_buffer) >=
      ab->audio->blocksize_read) {
      // Volume
      if (factor > 0)   {
          j = (gint16 *) (ab->audio->buffer + ab->audio->ptr_read_buffer);
          for (counter=0; counter<(ab->audio->blocksize_read/2); counter++) {
#if G_BYTE_ORDER == G_LITTLE_ENDIAN
            tmp = *j * factor;
            if (tmp > 32767) tmp = 32767;
            if (tmp < -32767) tmp = -32767;
            if (tmp >= 0) *j = (gint16) (tmp + 0.5);
            else *j = (gint16) (tmp - 0.5);
#elif G_BYTE_ORDER == G_BIG_ENDIAN
            tmp = GINT16_TO_BE(*j);
            tmp = tmp * factor;
            if (tmp > 32767) tmp = 32767;
            if (tmp < -32767) tmp = -32767;
            if (tmp >= 0) *j = GINT16_TO_LE((gint16) (tmp + 0.5));
            else *j = GINT16_TO_LE((gint16) (tmp - 0.5));
#else
#error "G_BYTE_ORDER should be big or little endian."
#endif
            j++;
          }
      }
      // plugins
      counter = 0;
      pn = (PluginEffectNode *) g_slist_nth_data(ab->list_plugins, counter);
      while (pn) {
          if (pn->active) {
            b = ab->audio->blocksize_read;
            pn->plugin->pefunction(pn->state, (ab->audio->buffer +
                        ab->audio->ptr_read_buffer), &b, AUDIO_CALL_IN);
          }
          counter++;
          pn = (PluginEffectNode *) g_slist_nth_data(ab->list_plugins, counter);
      }
      if (*mute != 1) {
          c = g_malloc(ab->audio->blocksize_read);
          memcpy(c, (ab->audio->buffer + ab->audio->ptr_read_buffer),
              ab->audio->blocksize_read);
          g_async_queue_push(ab->aqueue, c);
      }
      ab->audio->ptr_read_buffer += ab->audio->blocksize_read;
      if (ab->audio->ptr_write_buffer == ab->audio->ptr_read_buffer) {
         // return at the beginning
         ab->audio->ptr_write_buffer = 0;
```

```
            ab->audio->ptr_read_buffer = 0;
          }
          return 1;
        }
115     return 0;
}
/* ... */
```

En el caso del buffer de entrada, el funcionamiento es similar pero a la inversa, es decir se lee de la cola de audio un bloque de 1024 bytes -por ejemplo- y se gestiona en pequeños bloques para el codec de compresión.

Como se puede apreciar en el código, el buffer es circular, y nunca se va ha sobreescribir audio, ya que sólo se lee/escriben los datos bajo demanda, cuando no queda suficiente espacio o en el caso de la lectura, cuando no hay suficiente audio almacenado.

Hay que señalar también que para el cálculo del tamaño total del buffer se emplea el MCM (Mínimo Común Múltiplo). Se calcula el MCM de 2, del tamaño de bloque de audio del dispositivo -normalmente 1024 bytes- y del tamaño del bloque de audio que necesita el codec -generalmente menor de 512 bytes-. De esta forma, determinar cuando se ha llegado al final del buffer para volver a empezar por el principio es trivial: cuando el puntero de lectura (bloques de 512 bytes, por ejemplo) apunte al mismo lugar que el puntero de escritura (bloques de 1024 bytes). Con este método, se reducen gran cantidad de comprobaciones -muy difíciles si el tamaño de los buffers fuera fijo- a sólo una, línea 108.

Tanto el thread del buffer de entrada como el del buffer de salida, son totalmente independientes uno del otro, esa es la razón por la que el programa puede usar un codec para enviar audio y otro totalmente distinto (con distinto radio de compresión, tamaño de bloque de entrada, distinto tiempo de cómputo, etc.) para la salida.

6.7. Proceso RTP/RTP

Es el subsistema encargado de enviar, recibir y controlar los paquetes RTP que contienen el audio comprimido. Es un subsistema muy complejo, por lo que sólo se van a comentar las partes más importantes. Los ficheros de código fuente que engloba son: *io.c*, *io.h* y todos los del directorio *rtp*.

Consta de tres threads:

1. Lee los datos del doble buffer de audio -y éste, de una cola asíncrona de audio del

Procesador de Audio- y lo comprime con la clase *codec*, construye un paquete RTP y lo envía a la red con su payload correspondiente.

2. Recoge los paquetes RTP de la red, descomprime el audio -si puede-, lo envía al buffer de audio y éste lo escribe en las colas asíncronas productoras. Si resulta imposible descomprimir el audio, bien porque el paquete está dañado o porque el payload (contenido) ha cambiado, notifica esta circunstacia al thread 3, a través de otra cola de mensajes asíncronos (*qrtcp*).

3. Este thread es el encargado de la conexión RTCP, recibe los paquetes RTCP por el puerto inmediatamente superior al asignado a RTP, -siguiendo el estándar, el puerto es impar, ya que el puerto RTP debe ser par-. Asimismo, también controla los paquetes que le envían los dos threads (también en formato RTCP, pero estos locales, de tipo *APP*, subtipo *LOCAL_RTCP_APP*) a través de una cola asíncrona de mensajes (línea 10) y decide que hacer en función del tipo de paquete. También lleva toda la gestión de la sesión RTCP, tal como indica el estándar RFC 1889 (control de usuarios, envío de seguimiento online, envío de timeouts, *BYE's*, etc.). Otra de sus funciones es actualizar los datos relativos a la sesión RTP/RTCP en una estructura, para ofrecer una información detallada en la GUI. En el siguiente listado se muestra su implementación (fichero *io.c*).

```
/* ... */

while (asession->process) {
gint i;
gint32 ssrc_timeout = -1;
rtcp_app *app;

g_get_current_time(&time);
g_time_val_add(&time, 80000);
if ((p = g_async_queue_timed_pop(qrtcp, &time)) == NULL) {
g_mutex_lock(user_data.mutex);
   rtp_send_ctrl(asession->rtps, user_data.rtp_ts, NULL);
   rtp_update(asession->rtps);
   g_mutex_unlock(user_data.mutex);
} else {
    e = (RtpEvent *) p;
   switch (e->type) {
   case RX_SDES:
      // Warning controled ;-) (mutex)
      g_mutex_lock(rtpinfo->mutex);
      r = (rtcp_sdes_item *)e->data;
      sdes_print(asession->rtps, e->ssrc, r->type, rtpinfo);
      g_mutex_unlock(rtpinfo->mutex);
      break;
   case RX_BYE:
```

```
        g_mutex_lock(rtpinfo->mutex);
        if (rtpinfo->num_sources == 2) {
            asession->bye = TRUE;
            rtpinfo->bye = TRUE;
            asession->process = FALSE;
        }
        g_mutex_unlock(rtpinfo->mutex);
        break;
    case SOURCE_CREATED:
        g_mutex_lock(rtpinfo->mutex);
        rtpinfo->ssrcs[rtpinfo->num_sources] = e->ssrc;
        rtpinfo->num_sources = rtpinfo->num_sources + 1;
        g_mutex_unlock(rtpinfo->mutex);
        break;
    case SOURCE_DELETED:
        g_mutex_lock(rtpinfo->mutex);
        rtpinfo->num_sources = rtpinfo->num_sources - 1;
        for (i = 0; i < 10; i++) {
            if (rtpinfo->ssrcs[i] == e->ssrc) {
                rtpinfo->ssrcs[i] = 0;
                memset(rtpinfo->sdes_info[i][0], 0, 256);
                memset(rtpinfo->sdes_info[i][1], 0, 256);
                memset(rtpinfo->sdes_info[i][2], 0, 256);
                memset(rtpinfo->sdes_info[i][3], 0, 256);
                memset(rtpinfo->sdes_info[i][4], 0, 256);
            }
        }
        if (rtpinfo->num_sources == 1) rtpinfo->timeout = TRUE;
        g_mutex_unlock(rtpinfo->mutex);
        break;
    case RX_SR:
        break;
    case RX_RR:
        break;
    case RR_TIMEOUT:
        ssrc_timeout = e->ssrc;
        break;
    case RX_APP:
        app = (rtcp_app *) e->data;
        if ((app->ssrc == my_ssrc) && (app->subtype == LOCAL_RTCP_APP))    {
            // change payload
            // OK for the moment, becose ...
            g_mutex_lock(rtpinfo->mutex);
            rtpinfo->change_payload = TRUE;
            rtpinfo->payload = app->pt;
            g_mutex_unlock(rtpinfo->mutex);
        }
        break;
    default:
        break;
    }
    g_free(e->data);
    g_free(e->ts);
```

```
    g_free(e);
    g_mutex_lock(user_data.mutex);
    rtp_send_ctrl(asession->rtps, user_data.rtp_ts, NULL);
    rtp_update(asession->rtps);
    g_mutex_unlock(user_data.mutex);
}
/* ... */
```

Otra de las características importantes, es que el thread receptor RTP dispone de un algoritmo para controlar cuando el RTP remoto activa VAD (Voice Activity Detection) y hace transmisión discontinua (TD). Esta situación sería insalvable con otro tipo de arquitectura, sin tres threads independientes. Hay que tener en cuenta que si VAD está activado los paquetes llegan en secuencia, pero con distinto *timestamp*. A continuación se muestra la parte que controla estas situaciones:

```
/* ... */

enter = TRUE;
if (user_data->node_codec_rx != NULL) {
    if ((bytes = codec_decode(user_data->node_codec_rx->codec,
                    p->data, user_data->buffer_data_rx)) == PLUGIN_CODEC_NOTDECODE)
    {
      g_printerr("[IO] handler_rtp_event: el plugin seleccionado es incapaz
                    de descomprimir el audio.\n");
      enter = FALSE;
    }
} else {
   k = (guint16 *) p->data;
   j = (guint16 *) user_data->buffer_data_rx;
   for (i=0; i<(user_data->data_len_rx/2); i++) {
      *j = g_ntohs(*k);
      j++; k++;
   }
}
if (enter) {
   if (seq != 0) {
      while (++seq < p->seq) {
         if (control == 0) {
            control = write_audio_buffer(user_data->ab, 0,
                        user_data->mute_vol, user_data->buffer_data_rx);
          }
          timestamp += user_data->audio_ms_rx;
      }
   } else {
      seq = p->seq;
      timestamp = p->ts;
   }
   while ((timestamp += user_data->audio_ms_rx) < p->ts) {
      if (control == 0) {  /* escribe 0's */
         control = write_audio_buffer(user_data->ab, 0, user_data->mute_vol,
         user_data->buffer_data_rx);
```

```
      }
   }
   control = write_audio_buffer(user_data->ab, user_data->audio_gaim_rx,
40             user_data->mute_vol, user_data->buffer_data_rx);
} else {
   if (seq != 0) {
      while (++seq < p->seq) {
         if (control == 0) { /* escribe 0's */
45          control = write_audio_buffer(user_data->ab, 0, user_data->mute_vol,
                        user_data->buffer_data_rx);
          }
          timestamp += user_data->audio_ms_rx;
      }
50 } else {
      seq = p->seq;
      timestamp = p->ts;
   }
   while ((timestamp += user_data->audio_ms_rx) < p->ts) {
55    if (control == 0) {  /* escribe 0's */
         control = write_audio_buffer(user_data->ab, 0, user_data->mute_vol,
                     user_data->buffer_data_rx);
      }
   }
60 }
```

Finalmente se va a detallar el método ideado para cambiar el codec compresión de audio (de salida). Para ello se procede de la siguiente forma:

1. Se paran todos los (3) threads del subsistema RTP/RTCP, y se destruye el buffer de salida de audio, pero no la cola asíncrona a la que está conectado. Cuando se para todo el subsistema RTP/RTCP, no se indica nada al usuario remoto, él continúa enviando paquetes RTP y RTCP, puesto que los sockets no se han destruido, solo se ha salido eventualmente del proceso.

2. Se busca el codec que encaje con el payload requerido, si no se encuentra el codec, la sesión se destruye y se pierde irremediablemente.

3. Se crean un nuevo buffer de audio ajustado a las características requeridas por el codec y se le asigna la cola asíncrona original, antes liberada.

4. Finalmente, se crea otra vez el proceso RTP/RTCP y todo continúa funcionando de la misma forma, pero con un codec de salida distinto, ya que el de entrada (recibe el audio de la red) no se puede cambiar, no depende del usuario local.

6.8. Tratamiento de XML

Para acceder a los datos en formato XML de la agenda de contactos, se empleó la librería libxml-2. Provee multitud de funciones y modos de procesamiento (SAX, DOM, validación con el DTD, etc.). Este es el TDA que implementa el acceso a los datos XML:

```
/* ... addbook_xml.h */

struct _ItemUser
{
    gchar *name;
    gchar *user;
    gchar *host;
    guint16 port;
    gboolean ignore;
    gchar *description;
    gchar *photo;
};
typedef struct _ItemUser ItemUser;

/* ... */

#define XML_ROOT_NODE             "agenda"
#define XML_CONTACT               "contacto"
#define XML_CONTACT_DEFAULT       "default"
#define XML_CONTACT_IGNORE        "ignore"
#define XML_CONTACT_NAME          "nombre"
#define XML_CONTACT_USER          "usuario"
#define XML_CONTACT_HOST          "host"
#define XML_CONTACT_PORT          "puerto"
#define XML_CONTACT_DESCRIPTION   "descripcion"
#define XML_CONTACT_PHOTO         "foto"
#define XML_COMMENT               "Fichero de contactos, programa 'addressbook'"

xmlDocPtr xml_opendoc (gchar *docname, GError **error);
xmlDocPtr xml_newdoc ();
void xml_freedoc (xmlDocPtr doc);
gboolean xml_savedoc (xmlDocPtr doc, gchar *filename, GError **error);

gboolean xml_set_default_contact (ItemUser *user, xmlDocPtr doc);
gboolean xml_set_contact (ItemUser *user, xmlDocPtr doc);

GSList *xml_get_contact (xmlDocPtr doc, GError **error);
ItemUser *xml_get_default_contact (xmlDocPtr doc, GError **error);

ItemUser *xml_get_user (xmlNodePtr child);
gboolean is_user_ignored (xmlDocPtr doc, gchar *user, GError **error);
ItemUser *get_item_user (xmlDocPtr doc, gchar *user, GError **error);
ItemUser *get_user (gchar *username, gchar *db, GError **error);
void item_user_free(ItemUser *i);
```

Lo más llamativo en la implementación es que, en XML son considerados nodos las líneas en blanco y los comentarios, con lo que es necesario introducir comprobaciones para saber si el nodo que se está procesando es correcto o no. A continuación se muestra un trozo de código que ilustra esta situación:

```
/* ... addbook_xml.c */

gboolean xml_set_contact (ItemUser *user, xmlDocPtr doc)
{
    xmlNodePtr root;
    xmlNodePtr nodeuser;
    xmlNodePtr com;
    gchar *aux;

    root = xmlDocGetRootElement(doc);
    if (root == NULL) return FALSE;
    if (xmlStrcmp(root->name, (const xmlChar *) XML_ROOT_NODE)) return FALSE;
    com = xmlNewText("\n");
    xmlAddChild(root, com);
    nodeuser = xmlNewChild(root, NULL, XML_CONTACT, NULL);
    com = xmlNewText("\n\t");
    xmlAddChild(nodeuser, com);
    xmlSetProp(nodeuser, XML_CONTACT_IGNORE, (user->ignore) ? "TRUE" : "FALSE");
    xmlNewChild(nodeuser, NULL, XML_CONTACT_NAME, user->name);
    com = xmlNewText("\n\t");
    xmlAddChild(nodeuser, com);
    xmlNewChild(nodeuser, NULL, XML_CONTACT_USER, user->user);
    com = xmlNewText("\n\t");
    xmlAddChild(nodeuser, com);
    xmlNewChild(nodeuser, NULL, XML_CONTACT_HOST, user->host);
    com = xmlNewText("\n\t");
    xmlAddChild(nodeuser, com);
    aux = g_strdup_printf("%d", user->port);
    xmlNewChild(nodeuser, NULL, XML_CONTACT_PORT, aux);
    com = xmlNewText("\n\t");
    xmlAddChild(nodeuser, com);
    xmlNewChild(nodeuser, NULL, XML_CONTACT_DESCRIPTION, user->description);
    com = xmlNewText("\n\t");
    xmlAddChild(nodeuser, com);
    xmlNewChild(nodeuser, NULL, XML_CONTACT_PHOTO, user->photo);
    com = xmlNewText("\n");
    xmlAddChild(nodeuser, com);
    com = xmlNewText("\n");
    xmlAddChild(root, com);
    return TRUE;
}

/* ... */

ItemUser *xml_get_user (xmlNodePtr child)
{
    ItemUser *item;
```

```
    xmlNodePtr node;
    xmlChar *ig;

    if (child == NULL) return NULL;
    item = (ItemUser *) g_malloc0(sizeof(ItemUser));
    ig = xmlGetProp(child, (const xmlChar *) XML_CONTACT_IGNORE);
    if (ig == NULL) item->ignore = FALSE;
    else {
      if (!xmlStrcmp(ig, (const xmlChar *)"TRUE")) item->ignore = TRUE;
      else item->ignore = FALSE;
      xmlFree(ig);
    }
    node = child->xmlChildrenNode;
    while (node != NULL) {
      if ((!xmlStrcmp(node->name, (const xmlChar *)XML_CONTACT_NAME))) {
          item->name = xmlNodeGetContent(node);
      }
      if ((!xmlStrcmp(node->name, (const xmlChar *)XML_CONTACT_USER))) {
          item->user = xmlNodeGetContent(node);
      }
      if ((!xmlStrcmp(node->name, (const xmlChar *)XML_CONTACT_HOST))) {
          item->host = xmlNodeGetContent(node);
      }
      if ((!xmlStrcmp(node->name, (const xmlChar *)XML_CONTACT_PORT))) {
         ig = xmlNodeGetContent(node);
            item->port = atoi(ig);
         xmlFree(ig);
      }
      if ((!xmlStrcmp(node->name, (const xmlChar *)XML_CONTACT_DESCRIPTION))) {
          item->description = xmlNodeGetContent(node);
      }
      if ((!xmlStrcmp(node->name, (const xmlChar *)XML_CONTACT_PHOTO))) {
          item->photo = xmlNodeGetContent(node);
      }
      node = node->next;
   }
   return item;
}
```

La primera función muestra como hay que hacer para que el fichero XML resultante, quede indentado y pueda ser leído fácilmente por humanos. La segunda muestra el proceso inverso, la cantidad de comprobaciones necesarias para leer los datos significativos, saltándose los comentarios y/o líneas en blanco.

6.9. Proxy SSIP

En esta sección se cometarán las implementaciones más significativas desarrolladas en el proxy, aparte de las ya mencionadas en la sección de "Código Compartido", subsección "Negociación *SSIP* y *NOTIFY*" de este capítulo.

El **Proxy SSIP**, nada más ser ejecutado, se convierte en un demonio de sistema y crea dos procesos. Uno encargado de gestionar las notificaciones de alta/baja de usuarios (protocolo NOTIFY) y el otro de gestionar las negociaciones SSIP de los usuarios adscritos, para ello crea un canal de comunicación que enlaza transparentemente la conexión (en las especificaciones de requisitos y diseño se detallan más claramente los usos). El proceso de negociaciones SSIP crea a su vez otros procesos hijos, que gestionan una negociación de un usuario, ya que, la duración de una negociación SSIP puede durar mucho tiempo -hasta que venza un timeout o acepte/rechaze el otro usuario-. Mientras que en el proceso de notificación, las conexiones se atienden una por una y secuencialmente, ya que esta negociación es corta, sin esperas y sólo sucede una vez al principio y otra al final (no existe posibilidad de sobrecarga).

La parte más importante en el proxy ha sido el desarrollo del control de concurrencia entre procesos que acceden al fichero de base de datos de los usuarios del proxy. Para ello se implementó un método que usa el bloqueo de ficheros, como semáforo:

```
/* ... ssip_proxy.c */

void file_block (gint fd, gint mode)
{
    struct flock lk;

    lk.l_type = mode;
    lk.l_whence = SEEK_SET;
    lk.l_start = 0;
    lk.l_len = 0;
    lseek(fd, 0L, SEEK_SET);
    if (fcntl(fd, F_SETLKW, &lk)) {
      g_log(PROCESS_NOTIFY, G_LOG_LEVEL_CRITICAL,
      "Unable to set file block: %s", strerror(errno));
    }
}

/* ... */

gint add_user (gchar *filename, NtfPkt *pkt)
{
    gchar *from_user, *from_host, *des, *fileblock;
    guint16 fromport;
    gint ipmode, nada;
    gboolean value;
    ConfigFile *file;
    gchar *fileblock;
    gint fd_block;
    GError *tmp_error = NULL;

```

```
    file = cfgf_open_file(filename, &tmp_error);
    if (tmp_error != NULL) {
      g_log(PROCESS_NOTIFY, G_LOG_LEVEL_CRITICAL, "%s", tmp_error->message);
      g_error_free(tmp_error);
        return -1;
    }
    from_user = g_strndup(pkt->from_user, PKT_USERNAME_LEN);
    ipmode = g_ntohl(pkt->ipmode);
    if (ipmode != IPv6) {
      from_host = g_malloc0(INET_ADDRSTRLEN+1);
      inet_ntop(AF_INET, &(pkt->from_addr4), from_host, INET_ADDRSTRLEN);
      from_host[INET_ADDRSTRLEN] = '\0';
    }
#ifdef NET_HAVE_IPv6
    else {
      from_host = g_malloc0(INET6_ADDRSTRLEN+1);
      inet_ntop(AF_INET6, &(pkt->from_addr4), from_host, INET6_ADDRSTRLEN);
      from_host[INET6_ADDRSTRLEN] = '\0';
    }
#endif
    des = g_strndup(pkt->data, PKT_USERDES_LEN);
    fromport = g_ntohs(pkt->from_port);
    value = cfgf_read_int(file, from_user, "port", &nada);
    if (!value) {
      cfgf_write_string(file, from_user, "host", from_host);
      cfgf_write_int(file, from_user, "port", fromport);
      cfgf_write_string(file, from_user, "password", des);
      fileblock = g_strdup_printf("%s.lck", filename);
      fd_block = open(fileblock, O_RDWR | O_CREAT, 00700);
      file_block(fd_block, F_WRLCK);                    /* cerrojo de escritura */
      cfgf_write_file(file, filename, &tmp_error);
      cfgf_free(file);
      file_block(fd_block, F_UNLCK);                    /* fin cerrojo */
      g_free(fileblock);
      close(fd_block);
      value = TRUE;
    } else {
      cfgf_free(file);
      value = FALSE;
    }
    g_free(from_user);
    g_free(from_host);
    g_free(des);
    if (tmp_error != NULL) {
      g_log(PROCESS_NOTIFY, G_LOG_LEVEL_CRITICAL, "%s", tmp_error->message);
      g_error_free(tmp_error);
      return -1;
    }
    return value;
}

gint del_user (gchar *filename, NtfPkt *pkt)
{
```

```
    ConfigFile *file;
    GError *tmp_error = NULL;
    gchar *user, *des, *file_from_host, *file_des;
    gboolean value, value_host, value_pass;
    gint ipmode, fd_block;
    gchar *from_host, *fileblock;

    file = cfgf_open_file(filename, &tmp_error);
    if (tmp_error != NULL) {
      g_log(PROCESS_NOTIFY, G_LOG_LEVEL_CRITICAL, "%s", tmp_error->message);
      g_error_free(tmp_error);
      return -1;
    }
    user = g_strndup(pkt->from_user, PKT_USERNAME_LEN);
    des = g_strndup(pkt->data, PKT_USERDES_LEN);
    ipmode = g_ntohl(pkt->ipmode);
    if (pkt->ipmode != IPv6) {
      from_host = g_malloc0(INET_ADDRSTRLEN+1);
      inet_ntop(AF_INET, &(pkt->from_addr4), from_host, INET_ADDRSTRLEN);
        from_host[INET_ADDRSTRLEN] = '\0';
    }
#ifdef NET_HAVE_IPv6
    else {
      from_host = g_malloc0(INET6_ADDRSTRLEN+1);
      inet_ntop(AF_INET6, &(pkt->from_addr4), from_host, INET6_ADDRSTRLEN);
      from_host[INET6_ADDRSTRLEN] = '\0';
    }
#endif
    value = TRUE;
    value &= value_host = cfgf_read_string(file, user, "host", &file_from_host);
    value &= value_pass = cfgf_read_string(file, user, "password", &file_des);
    des[PKT_USERDES_LEN-1] = '\0';
    if ((value) && (!strcmp(file_des, des)) &&
      (!strcmp(file_from_host, from_host))) {
      cfgf_remove_key(file, user, "host");
      cfgf_remove_key(file, user, "port");
      cfgf_remove_key(file, user, "password");
      fileblock = g_strdup_printf("%s.lck", filename);
      fd_block = open(fileblock, O_RDWR | O_CREAT, 00700);
      file_block(fd_block, F_WRLCK);                    /* cerrojo de escritura */
      cfgf_write_file(file, filename, &tmp_error);
      cfgf_free(file);
      file_block(fd_block, F_UNLCK);                    /* fin cerrojo */
      g_free(fileblock);
      close(fd_block);
      g_free(file_from_host);
      g_free(file_des);
      value = TRUE;
    } else {
      if (value_host) g_free(file_from_host);
      if (value_pass) g_free(file_des);
      cfgf_free(file);
      value = FALSE;
```

```
    }
    g_free(user);
    g_free(des);
    g_free(from_host);
    if (tmp_error != NULL) {
      g_log(PROCESS_NOTIFY, G_LOG_LEVEL_CRITICAL, "%s", tmp_error->message);
      g_error_free(tmp_error);
      return -1;
    }
    return value;
}
/* ... */

   fileblock = g_strdup_printf("%s.lck", filename);
   fd_block = open(fileblock, O_RDWR | O_CREAT, 00700);
   file_block(fd_block, F_RDLCK);                    /* cerrojo de lectura */
   file = cfgf_open_file(filename, &tmp_error);
   file_block(fd_block, F_UNLCK);                    /* fin del cerrojo */
   g_free(fileblock);
   close(fd_block);
/* ... */
```

Hay ciertos aspectos de las funciones anteriores que merecen ser comentados:

Control de concurrencia. Como se aprecia en las líneas 60, 123 y 152 antes que un proceso acceda al fichero BD, bloquea otro fichero, con el mismo nombre pero con extensión ".lck". Este fichero se crea si no existe. Cada proceso que accede al fichero de BD primero comprueba si hay cerrojos establecidos sobre el segundo. Hay cerrojos de lectura: puede haber varios sobre el fichero -línea 152- (admite concurrencia en la lectura) y cerrojos de escritura -líneas 60 y 123- que son totalmente excluyentes. Si un proceso bloquea el fichero para escritura, cualquier otro proceso que acceda al fichero se quedará suspendido hasta que el primero quite el cerrojo. Los cerrojos están implementados con la llamada al sistema *fcntl* como se aprecia en la línea 12.

Registro exhaustivo de incidencias. Como se aprecia en las líneas 13, 33, 75, 93 y 142, con varios niveles de prioridad. Cualquier servidor crítico como es el caso, debe "logear" todas las incidencias para poder determinar el reponsable de posibles acciones.

Compilación condicional para IPv6. De esta forma se permite que el software pueda ser compilado con soporte para IPv6.

Otra consideración muy importante es el control del número de procesos hijos, para poder limitar el número de hijos se usan señales. Cada vez que se crea un hijo se incrementa un contador (línea 72), cuando el hijo termina, manda una señal (SIGCHLD) al padre y este hace un *wait* (así evita que queden processos "zombis") y decrementa ese contador

(línea 9). También se usan señales para que el administrador pueda interrumpir el proceso de notificación, para ello debe mandar la seña SIGUSR1 a dicho proceso (línea 11).

```
/* ... ssip_proxy.c */

void handler_signal_manage (int sig)
{
    g_log(PROCESS_NOTIFY, G_LOG_LEVEL_INFO, "Signal received (%d)", sig);
    switch (sig) {
   case SIGCHLD:
      wait(0);
      childcount--;
      break;
    case SIGUSR1:
      go_manage_process = FALSE;
      go_notify_process = FALSE;
      while (childcount > 0) wait(0);
      break;
    default:
      break;
    }
}
/* ... */

   /* Instalación de los manejadores de señal */

    g_log(PROCESS_MANAGE, G_LOG_LEVEL_INFO, "PROCESS_MANAGE --- INIT ---,
          pid=%d at port %d", getpid(), port);
    action.sa_handler = handler_signal_manage;
    sigemptyset(&action.sa_mask);
    action.sa_flags = SA_NOCLDSTOP;
    if ((sigaction(SIGUSR1, &action, NULL)) < 0) {
      g_log(PROCESS_MANAGE, G_LOG_LEVEL_CRITICAL, "%s", strerror(errno));
      kill(ch_pid, SIGUSR1);
      wait(0);
      return FALSE;
    }
    if ((sigaction(SIGUSR2, &action, NULL)) < 0) {
      g_log(PROCESS_MANAGE, G_LOG_LEVEL_CRITICAL, "%s", strerror(errno));
      kill(ch_pid, SIGUSR1);
      wait(0);
      return FALSE;
    }

/* ... */

   while (childcount >= num_forks_manager)    {
      g_log(PROCESS_MANAGE, G_LOG_LEVEL_DEBUG, "Limits FORKS (%d)", childcount);
      sleep(1);
   }
/* ... */

   if ((pid = fork()) == 0) {
```

```
      pid = getpid();
      g_log(PROCESS_MANAGE, G_LOG_LEVEL_INFO, "FORK --- INIT --- MANAGE_PROCESS,
          pid=%d --- remote host: %s", pid, sacpt->addr);
      tcp_exit(Sc, &tmp_error);
      if (tmp_error != NULL) {
          g_log(PROCESS_MANAGE, G_LOG_LEVEL_WARNING, "FORK --- (pid=%d) --- %s",
            pid, tmp_error->message);
          g_clear_error(&tmp_error);
      }
      protocol_manager(sacpt, filename);
      g_log(PROCESS_MANAGE, G_LOG_LEVEL_INFO, "FORK --- EXIT --- MANAGE_PROCESS,
          pid=%d --- remote host: %s", pid, sacpt->addr);
      tcp_exit(sacpt, &tmp_error);
      if (tmp_error != NULL) {
          g_log(PROCESS_MANAGE, G_LOG_LEVEL_WARNING, "FORK --- (pid=%d) ---
          Unable close socket: %s", pid, tmp_error->message);
          g_clear_error(&tmp_error);
      }
      kill(fpid, SIGCHLD);
      exit(1);
   } else {
      if (pid > 0) childcount++;
      else g_log(PROCESS_MANAGE, G_LOG_LEVEL_CRITICAL, "Unable to fork:
          %s (remote host: %s)", strerror(errno), sacpt->addr);
/* ... */
```

6.10. Otras consideraciones

Como se puede apreciar a lo largo de todo el código mostrado, todos los tipos de datos son los ofrecidos por Glib, de esa forma se gana en portabilidad entre plataformas. Además todo el tratamiento de errores se realiza con *GError*, que ofrece un interfaz cómodo para lidiar con esos inesperados "amigos": funciones de creación, propagación, copia, etc. Todos los plugins usan GModule para exportar las funciones (símbolos) al programa principal. La interfaz gráfica, implementada íntegramente con GTK+, ofrece portabilidad hasta para sistemas *MS Windows*.

Capítulo 7

Pruebas, resultados y rendimiento

Índice General

Para la realización de todas las pruebas, el programa se ejecutó en dos máquinas con estas características:

CPU 1 : Athlon 700 Mhz.

Memoria 1 : 256 MB.

SO 1 : Debian GNU/Linux "Sarge" (testing).

CPU 2 : Pentium II 350 Mhz.

Memoria 2 : 192 MB.

SO 2 : Debian GNU/Linux "Sarge" (testing).

Red : IPv4 (local, a través de switch): 10 MB/s.

Retardo medio entre paquetes de la red: 0.760 ms.

El programa estaba compilado para plataformas I386, es decir sin ningún tipo de optimización y con símbolos de depurado: *-g -ggdb*. El formato de audio tratado internamente en el programa es:

Tamaño : 2 bytes (16 bits).

Ordenamiento bytes : LITTLE ENDIAN.

Frecuencia : 8000 Hz.

Canales : monoaural (1 canal).

Dichas pruebas en la LAN han sido realizadas sin ningún tipo de obstáculo al tráfico, como cortafuegos, y sin la presencia del proxy SSIP.

7.1. Pruebas y resultados del software

Debido al método evolutivo de desarrollo del software, se fueron realizando pruebas secuencialmente, a medida que se añadían características, con lo que las pruebas de requerimiento apenas existieron. Muchas pruebas se realizaron con programas de apoyo implementados específicamente para esos menesteres (pruebas de caja blanca). El objetivo principal era comprobar que no se iban acumulando fallos en los subsistemas que se iban desarrollando y perfeccionando. Sin embargo, una vez finalizado el proyecto, se realizaron una serie de pruebas que se detallan a continuación (pruebas de caja negra):

Pruebas de configuración y arranque del sistema

Propósito Comprobar que el software desarrollado es robusto frente a operaciones malintencionadas ó erróneas en el momento del arranque y configuración. Entre estas acciones se pueden incluir: introdución incorrecta de parametros en la interfaz, inexistencia de un fichero de configuración e incorrecciones sintácticas ó léxicas en dicho fichero de configuración, etc. Dentro de estas pruebas, también se incluyen las del subsistema de procesamiento de la agenda de contactos en formato XML.

Tratar de introducir direcciones IP imposibles (con algunos de sus campos superiores a 255) o con tres ó cinco número decimales. También probar a usar puertos superiores a 65.535 ó puertos negativos. Los posibles errores que se producen son: una entrada no reconocida (error sintáctico), el valor asignado a otra no se corresponde con lo esperado (error léxico) y por último la imposibilidad de generar de forma interna un parámetro esencial para el funcionamiento que no ha sido especificado por el usuario (error al generar un parámetro).

Comprobación del subsistema RTP/RTCP

Propósito El propósito de esta prueba consiste en comprobar que el subsistema RTP y RTCP cumple el estándar y es capaz de resistir paquetes mal formados o incorrectos.

Para ello se usó la herramienta que analiza el tráfico de red *etherape* y es capaz de reconocer paquetes RTP y localizar todos sus campos. Para el envío incorrecto de paquetes RTP mal formados, se empleó el comando *nc* (netcat) que permite enviar datos arbitrarios a un puerto especificado.

Comprobación del subsistema SSIP/NOTIFY

Propósito comprobar que el subsistema SSIP y NOTIFY es robusto, resiste paquetes mal formados o incorrectos y con conexiones erróneas.

También se usó la herramienta *etherape* para analizar el tráfico de red junto con *netcat* para probar si realmente funcionan los timeouts y las comprobaciones en todos los campos del paquete.

Pruebas de estrés

Propósito Someter al programa a un test de funcionamiento continuo.

Consiste en poner a funcionar el programa durante unas 6 horas seguidas para comprobar con ayuda de *memprof*, si quedan regiones de memoria perdidas. Otra prueba consistió en modificar el código fuente para que creara sesiones RTP indefinidamente. La primera prueba permitió localizar zonas de memoria perdidas y corregir los fallos. La segunda permitió comprobar la robusted del programa, en el PC 2, consiguió crear 197 threads concurrentes.

Pruebas al Proxy SSIP

Propósito Comprobar que el proxy no crea procesos indefinidamente y responde a las especificaciones. También pruebas de estrés.

Como gran parte del código del proxy está reutilizado, se procedió a comprobar que es capaz de limitar el número de procesos hijo y que registra cualquier incidencia mediante *syslog*. Para ello se colocó el número máximo de hijos en 1 y se intentó lanzar una segunda negociación que fue rechazada y "logeada" inmediatamente. También se lanzó un bucle de conexiones erróneas que el proxy superó perfectamente.

7.2. Rendimiento

El rendimiento del programa depende de varios factores, los más importantes son

Carga del sistema . En sistemas multicast, como son los de tipo Unix, todos los procesos compiten en "igualdad" de condiciones para conseguir la CPU. Como esta aplicación tiene un componente de tiempo real, si no consigue la CPU cada cierto tiempo se pueden perder datos, ya que el sonido es un flujo constante.

MTU de la red . Si se envían paquetes RTP muy pequeños (comprimidos), la transmisión no resultará óptima, ya que, el bloque de datos, al ir descendiendo por la pila de protocolos, se van añadiendo cabeceras. Si esas cabeceras ocupan una porción significativa del paquete, no se estará transmitiendo óptimamente, ya que habrá mucha fragmentación en la red, con paquetes muy pequeños.

Tamaño del bloque de audio del dispositivo . Si el tamaño es muy pequeño, el sistema necesitará leer más asiduamente bloques de audio, mientras que con un tamaño muy grande, se puede permitir más relajación. Las ventajas de un tamaño de bloque pequeño, son que se reduce el retardo de la conversación (lag) a cambio de más uso de CPU, con un tamaño grande, se incrementa la latencia (lag) resultando molesto si dura más de 3/4 de segundos, pero, no se necesita tanta CPU.

En general, el rendimiento de la aplicación en condiciones normales es bueno, el uso de la CPU no llega al 1 % -para una sesión y con el plugin de compresión ADPCM-. El rendimiento global depende mucho de los plugins que se empleen, si necesitan mucha CPU, se estará reduciendo el tiempo a los demás threads concurrentes, con lo que puede suceder que se pierdan frames de audio. En general el rendimiento computacional de todo el programa en un caso normal está entre *O(log n)* y *O(n)*.

Capítulo 8

Conclusiones

Índice General

Durante el desarrollo de esta memoria se han expuesto todos los detalles acerca de la creación de este "Servicio de talk verbal". Aquí se muestran las comparativas, posibles mejoras o ampliaciones y las conclusiones extraídas de la realización del proyecto.

8.1. Concordancia entre resultados y objetivos

Está bastante claro que al final, el proyecto ha superado la propuesta inicial. Mientras que inicialmente, se proponía construir simplemente un servicio de talk verbal para Linux y, como posible ampliación, el desarrollo de plugins de compresión de audio. El proyecto final tiene como características más relevantes:

- Soporte de multisesión. El usuario del programa puede conectar con varias personas simultáneamente, sin necesidad de redes multicast. Incluso tiene capacidad para realizar coloquios a tres o más bandas.
- Soporte completo de IPv6. El programa está perfectamente adaptado para funcionar con la próxima generación de IP. También soporta redes multicast, tanto en IPv4 como en IPv6.
- Arquitectura totalmente abierta. Si inicialmente se barajaba la posibilidad de la implementación de plugins de compresisón de audio. Esta característica ha sido ampliamente superada, ahora no sólo soporta plugins de compresión de audio, también de efectos: para grabar conversaciones, modificar la voz, etc. y de E/S de audio: para OSS, ALSA, para Windows con su sistema de audio, etc.
- Portabilidad. El empleo de determinadas librerías, implica que el programa no sólo funciona en Linux, sino que en otros sistemas compatibles Unix (POXIX). Incluso puede ser portado fácil e inmediatamente a plataformas Windows cambiando el API de sockets por WinSock.
- Comunicación full-duplex. Al igual que ocurre con el teléfono, admite la comunicación bidireccional simultánea.
- Sistema de negociación de inicio de sesión. Permite que los usuarios puedan seleccionar los codecs que van a ser usados en la comunicación y otros parámetros.
- Soporte para cambio dinámico de plugin de compresión, el usuario tiene libertad para cambiar el algoritmo de compresión que usa en la comunicación, de este modo se puede adaptar a nuevas características de la red: mayor o menor consumo de ancho de banda, retardos, etc

- GUI intuitiva y rápida. Cada persona puede tener asignada una foto, de modo que resulta más fácil conocer de un vistazo quién llama. Además dispone de una agenda que almacena los contactos para llamarlos con sólo dos "clicks" de ratón. El formato de la agenda se almacena en XML, de esta forma puede ser exportado fácilmente a/desde otras aplicaciones.
- Aparte de los controles de audio tradicionales -volumen y ganancia- dispone de VAD (Voice Activity Detection) y CNG (Comfort Noise Generation). El VAD permite ahorrar ancho de banda en la transmisión, al eliminar los silencios. El CNG, hace creer al usuario que la comunicación está activa cuando se usa VAD, así se evitan expresiones del tipo "me escuchas", "sigues ahí", etc. También puede cortar el audio del micrófono y/o altavoz para un usuario en particular o para todos. Dispone también de un amplificador de audio para cada usuario.
- Totalmente compatible con el estándar RTP/RTCP (RFC 1889). Esto garantiza la interoperabilidad con otros programas. Además soporta el cifrado de la conexión RTP/RTCP con el algoritmo DES.
- Proxy. Se ha desarrollado un programa adiccional llamado **Proxy SSIP** que permite que el programa principal sea usado en subredes destrás de NAT. También permite gestionar a muchos usuarios simultáneamente.

Como se puede observar, todas estas características superan ampliamente los objetivos de la propuesta inicial, si bien el autor todavía no está conforme y planea aumentar esta lista de características, como se mostrará en la sección 8.3 "Mejoras y ampliaciones".

8.2. Comparativa con otros programas

Se va a comparar el software desarrollado con las aplicaciones analizadas inicialmente en el capítulo 2:

Linphone. Características de *Linphone* comparadas con **Asubío**:

- Ambos están inicialmente desarrollados para linux u otras plataformas compatibles *Unix*.
- GUI intuitivas y fáciles de usar. *Linphone* se ha desarrollado con GNOME, mientras que **Asubío**se ha desarrollado con GTK+ (a más bajo nivel). *Linphone* dispone de una aplicación para consola *linphonec*, por lo que no necesita el sistema gráfico para funcionar. También dispone -en la última versión- de una agenda de contactos, al igual que **Asubío**, si bien la agenda no está en formato XML.

- *Linphone* incluye una gran variedad de codecs compresores de audio: ADPCM. GSM, Speex, etc. pero todos están integrados en el código, es decir no dispone de plugins.
- Utiliza el protocolo SIP para negociar el inicio de sesión. **Asubío** usa un protocolo implementado especialmente: SSIP. La ventaja es que SIP está estandarizado.
- Ambos proyectos son de software libre, liberados bajo los términos de la *GPL*.
- No tiene capacidad de multisesión, VAD, CNG, control de ganancia de audio para cada usuario. Tampoco puede mostrar una foto del usuario que llama o está conversando.
- Ambos programas tienen la posibilidad de cambio dinámico de codec de compresión de audio en medio de una conversación.
- *Linphone* tiene una aquitectura cerrada, no dispone de plugins. **Asubío** puede cargar plugins para tres funciones distintas.
- Ambos están programados en C, pero *Linphone* usa más exhativamente OO.
- *Linphone* no soporta cifrado de RTP ni el protocolo RTCP.

La ventaja más significativa de *Linphone* respecto **Asubío** es el uso de SIP como protocolo de inicio de sesión.

Speak Freely.

- *Speak Freely* es multiplataforma, existen versiones tanto para sistemas *Unix* como *Windows*. **Asubío** sólo funciona -de momento- en plataformas *Unix*.
- *Speak Freely* tiene interoperabilidad con ICQ, un servicio de mensajería instantánea.
- La GUI de *Speak Freely* está realizada con wxWindows, mientras que **Asubío** se ha desarrollado con GTK+.
- Ambos permiten charlar con varias personas a la vez, sin necesidad de estar en redes *multicasting*.
- *Speak Freely* Permite grabar mensajes de voz que informen a la persona llamante de que no está disponible.
- Ambos soportan cifrado RTP, pero *Speak Freely* implementa varios algoritmos de cifrado: DES, IDEA, PGP, Blowfish, etc.

- Aparte de RTP, implementa el protocolo *VAT* (Visual Audio Tool), un protocolo muy simple de streaming de audio en tiempo real. Esta característica no es significativa actualmente, ya que es protocolo es antiguo y no está estandarizado.
- *Speak Freely* soporta estos codecs compresores de audio: DVI4, GSM, L16, LPC, PCMA y PCMU, pero están integrados dentro del programa, no tiene plugins.
- Ambos programas tienen la posibilidad de cambio dinámico de codec de compresión de audio, en medio de una conversación activa.
- Ambos proyectos son de software libre, liberados bajo los términos de la *GPL*.
- Ambos están programados en C.
- No tiene capacidad de: VAD, CNG, control de ganancia de audio para cada usuario. Tampoco dispone de agenda de contactos, ni de plugins de ningún tipo.

Speak Freely no tiene ninguna ventaja significativa que mejore las prestaciones de **Asubío**.

RAT - Robust Audio Tool.

- Multiplataforma, FreeBSD, HP-UX, IRIX, Linux, NetBSD, Solaris, SunOS y Windows 95/NT. **Asubío** sólo "corre" -por ahora- en sistemas compatibles con POSIX.
- Tiene una GUI poco actractiva y sin agenda de contactos.
- Tanto *RAT* como **Asubío** incorporan técnicas de VAD.
- Sporta multiconferencia, pero necesita redes *multicasting*, mientras que con **Asubío** no es necesario disponer de una red *multicast* para multisesión.
- Completamente compatible con el estándar RTP/RTCP, puede incluso enviar audio redundante para contrarestar pérdidas de paquetes.
- Soporta cifrado de sesiones RTP/RTCP con DES, al igual que **Asubío**
- Soporta varios codecs compresores de audio: GSM, ADPCM (G.726), G.711 y LPC. Están integrados dentro del programa, no tiene plugins.
- Es software libre, liberado bajo los términos de la *GPL*.

La característica más significativa de *RAT* es que es el único programa que integra sistemas de envío de audio redundante, para evitar pérdidas de paquetes. Si bien esas ténicas no están estandarizadas, todavía están en fase "draft" o borrador, es decir, están

en fase de estudio y pueden sufrir ligeras modificaciones.

También hay que comentar que ninguno de los programas analizados (para Linux), disponen de la implementación de un proxy, capaz de realizar las gestiones de negociación de inicio de sesión de una forma fácil y cómoda para un grupo de usuarios o para una subred detrás de NAT.

En general, las comparaciones, se resumen de esta forma: **Asubío** es una mezcla de características de varios programas, pero, también implementa otras que ninguno de los programas analizados posee. Por ejemplo su arquitectura modular que le permite cargar plugins que pueden extender su funcionalidad ilimitadamente. Asimismo, su comunicación mediante colas asíncronas entre threads, es un sistema que le aporta gran vesatilidad.

Pero, también hay que tener en cuenta que, mientras que este proyecto tiene un tiempo de realización de menos de 9 meses, todos los programas anteriores cuentan con años de desarrollo, junto con muchos más desarrolladores y testeadores.

8.3. Mejoras y ampliaciones

La versión actual de todo el software desarrollado se ha establecido en 0,1,0. Esto puede dar una idea inicial de hasta donde pretende llegar, porque, aparte de ser completamente funcional, se pretende seguir ampliando el programa con ayuda de desarrolladores del ancho mundo, gracias a la filosofía del software libre. Para ello existe una cuenta de desarrollo en *http://asubio.sourceforge.net*.

A continuación se presenta una lista con las mejoras o ampliaciones más prioritarias que deberían ser aplicadas al proyecto:

1. Desarrollo de nuevos plugins de compresión de audio, así se podría elegir más variedad de algoritmos con distinto ancho de banda. También es necesario implementar nuevos *InOut Plugin* que den soporte a ALSA, por ejemplo.

2. Desarrollar un agente de usuario interno con el protocolo SIP, sólo con esta mejora, se podría pasar a la versión 0,2,0, ya que supone un salto muy importante tanto en compatibilidad con otros programas como en cumplimiento de estándares. Por contra, la implementación de H.323 no corre tanta prioridad, pudiendo ser añadida en versiones superiores a la 1,0,0, ya que este protocolo es cerrado y rígido.

3. Implemetar varios sistemas de reparación de audio, lo que permitiría que, aunque se pierdan paquetes RTP, el usuario no note esas pérdidas. Existen varias estrategias para resolver el problema, detalladas en distintos borradores de internet:

   Envío redundante de audio. Cada vez que se envía un paquete, se aprobecha para enviar los n anteriores, conjuntamente. De esta forma, si se pierde algún paquete, sólo es necesario esperar al siguiente, el cual contendrá el audio para el instante actual y para n instantes anteriores. El problema se produce cuando se pierden n paquetes o más, no existen posibilidades de reparación. El inconveniente de este sistema es que multiplica por n el ancho de banda requerido y, en el peor de los casos, se puede producir un retardo de varios segundos en el audio recibido.

   Entrelazado de audio Una vez formado un bloque de audio, se divide en n trozos, forma un nuevo bloque compuesto por n trozos ordenados secuencialmente de los paquetes originales, y se envía ese bloque compuesto. Para recomponer un bloque, es necesario esperar la llegada de n. La ventaja de este sistema es que no aumenta el ancho de banda consumido, pero la conversación sufre un retardo constante de varios segundos, debido a la necesidad de esperar la llegada de n bloques. Si se pierde un bloque el usuario no notará nada, ya que afectará mínimamente a los n bloques sucesivos.

4. Añadir portabilidad a otras plataformas, como Windows, para ello habrá que implementar un plugin que acceda al sonido en esa plataforma y añadir soporte para WinSock.

8.4. Conclusiones del proyecto

Como se puede apreciar las tecnologías aplicadas funcionan de forma bastante razonable. La implementación del prototipo demostró que era perfectamente factible llevar a la práctica el análisis y diseño propuestos y que el objetivo que se trató de perseguir desde un primer momento podía ser cumplido.

Las pruebas realizadas con el prototipo y su comparativa con otros proyectos similares han demostrado que en determinadas características iguala o incluso supera a sus competidores. Por otra parte, que este proyecto pueda comportarse tan bien o en ocasiones incluso mejor que algunos programas desarrollados por empresas, demuestra que las decisiones en cuanto a eficiencia fueron, aunque seguramente mejorables, bastante buenas.

El diseño de la GUI y de la ventana de configuración, basados en pestañas, permiten añadir nuevas funcionalidades al interfaz gráfico, sin apenas modificar el diseño. Sólo basta con añadir nuevas pestañas que configuren nuevos parámetros.

El modelo multi-hilo utilizado reacciona y se comporta bastante bien en situaciones de carga elevada si bien llega un límite (de sesiones) a partir del cual el rendindimiento se reduce, pero este problema puede resolverse simplemente mejorando el hardware sobre el que corre el programa. Además, el diseño utilizado para el acceso al audio mediante una capa de abstracción del dispositivo de audio, ha mostrado que supera al enfoque utilizado por otros programas que: o bien no soportan multisesión o bien no soportan el cambio dinámico de codec de compresión de audio (plugin). El sistema de plugins permite extender la funcionalidad de todo el sistema de forma fácil, rápida y cómoda. También el uso de estándares como RTP garantiza la compatibilidad con otros programas. La posibilidad de uso de IPv6, junto con el soporte para redes multicast, garantiza larga vida a este software, ya está adaptado para el futuro próximo.

El modelo flexible de implementación, mezclando técnicas de OO con programación tradicional, basada en TDA's ha mostrado, una gran flexibilidad a la hora de desarrollar ciertos subsistemas, sin comprometer la eficiencia y el rendimiento. Además ese modelo permitirá implementar fácilmente nuevas características. Por ejemplo, para añadir el ansiado soporte de SIP, basta con modificar sólo dos funciones: una para la negociación de paquetes de entrada y otra para los de salida.

Pronto habrá más información en la página web oficial del proyecto: *http://asubio.sf.net*

Bibliografía

[1] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. 1990.

[2] Internet Engineering Task Force. Ietf. http://www.ietf.org, 2000.

[3] Google. Google. http://www.google.com, 2000.

[4] M. Handley and V. Jacobson. Sdp: Session description protocol. http://www.faqs.org/rfcs/rfc2327.html, 1998.

[5] Jesús Bobadilla Sancho y Pedro Gómez Vilda Jesús Bernal Bermúdez. *Reconocimiento de voz y fonética acústica*. Ra-Ma, 2000.

[6] C. Liu. Multimedia over IP: RSVP, RTP, RTCP, RTSP. *Technical Report, Ohio State University*, January 1998.

[7] UCL Multimedia. Rat. http://www-mice.cs.ucl.ac.uk/multimedia/software/rat/, 2000.

[8] Bradford Nichols, Bick Buttlar, and Jackie Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1996.

[9] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. Rtp: A transport protocol for real-time applications. http://www.faqs.org/rfcs/rfc1889.html, 1996.

[10] Richard W Stevens. *UNIX Network Programming*. Software Series. Prentice Hall PTR, 1990.

[11] Open Sound System. Oss. http://www.opensound.com, 2000.

[12] International Telecommunication Union. Itu. http://www.itu.int/home/, 2000.

[13] Kurt Wall. *Programación en Linux con ejemplos*. McGraw-Hill, 1 edition, 2000.

[14] José Manuel Huidobro Moya y David Roldán Martínez. *Integración de voz y datos*. McGraw-Hill, 1 edition, 2003.

[15] Jonathan Davidson y James Peters. *Fundamentos de voz sobre IP*. Cisco Press, 2001.