Implementaciones de VoIP basadas en SIP: un estudio utilizando Asterisk

Instituto de Ingeniería Eléctrica
Facultad de Ingeniería - UDELAR
Agosto de 2005

Tutores:
Ing. Gabriel Gómez
Ing. Luis Vázquez

María Eugenia Caetano |
Gonzalo Mateos |
Federico Morales |

RESUMEN

Se estudian e implementan estructuras de Voz sobre IP (VoIP) sobre redes de área local (LAN), utilizando las funcionalidades que ofrece Asterisk, la PBX Open Source.

Se profundiza el estudio sobre el canal SIP de Asterisk, de forma de entender su funcionamiento y así poder extender las funcionalidades del sistema. En este contexto se desarrolla el soporte de IPv6 para dicho canal.

Como práctica de diseño de un sistema real de VoIP basado en SIP y Asterisk, se presenta una solución completa en hardware y configuración del servidor para cumplir con los requerimientos de telefonía en un caso real: el Instituto de Ingeniería Eléctrica de la Facultad de Ingeniería.

AGRADECIMIENTOS

Queremos agradecer a:

Teledata S.A. y Softnet Logical Uruguay S.A., por el préstamo desinteresado de equipamiento, sin el cual no hubiéramos podido realizar las distintas pruebas que permitieron el desarrollo del proyecto.

Juan, por el diseño de la carátula.

ABB, por facilitarnos los materiales para la impresión de la documentación.

Los compañeros del I.I.E., y puntualmente a Pablo Belzarena, Pedro Casas, Eduardo Cota, Alicia Fernández, Rafael Grompone, Laura Landín, Federico Larroca, Federico Lecumberry y Álvaro Valdés, por estar siempre dispuestos a responder nuestras dudas y por su apoyo moral.

Nuestros tutores Gabriel y Luis, por haber sido siempre abiertos a nuestras ideas y colaborar aportando las suyas, además de asesorarnos en diversos aspectos técnicos gracias a su amplia experiencia y conocimiento en esta área.

Nuestras familias y amigos, por alentarnos continuamente a terminar con el proyecto lo antes posible y soportarnos durante horas a lo largo de varias jornadas de trabajo.

Y finalmente, a todos aquellos que nos olvidamos de nombrar en esta lista.

TABLA DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CONTENIDO DEL CD

El CD se encuentra en la contratapa de la presente documentación y contiene:

1. Versión electrónica de la presente documentación.

Nombre de archivo	Directorio
documentacion.pdf	\documentacion\

2. Archivos fuente de las modificaciones realizadas al sistema (por más información consultar el Capítulo 6). También se adjuntan los archivos fuente de las diferentes versiones de Asterisk donde se aplican dichas modificaciones.

Nombre de archivo	Directorio
asterisk-1.0.8-ipv6.patch	\src\patches\
asterisk-1.0.9-ipv6.patch	\src\patches\
asterisk-cvs220705-ipv6.patch	\src\patches\
asterisk-1.0.8.tar.gz	\src\1.0.8\
libpri-1.0.8.tar.gz	\src\1.0.8\
zaptel-1.0.8.tar.gz	\src\1.0.8\
asterisk-1.0.9.tar.gz	\src\1.0.9\
libpri-1.0.9.tar.gz	\src\1.0.9\
zaptel-1.0.9.tar.gz	\src\1.0.9\
asterisk-cvs220705.tar.gz	\src\cvs\
libpri-cvs220705.tar.gz	\src\cvs\
zaptel-cvs220705.tar.gz	\src\cvs\

3. Archivos de configuración de Asterisk relativos al sistema diseñado como sustituto a la PBX del I.I.E. (por más información consultar el Capítulo 7).

Nombre de archivo	Directorio
`sip.conf`	`\diseno_sistema_IIE\confs\`
`zapata.conf`	`\diseno_sistema_IIE\confs\`
`zaptel.conf`	`\diseno_sistema_IIE\confs\`
`voicemail.conf`	`\diseno_sistema_IIE\confs\`
`extensions.conf`	`\diseno_sistema_IIE\confs\`
`features.conf`	`\diseno_sistema_IIE\confs\`
`meetme.conf`	`\diseno_sistema_IIE\confs\`

4. Archivo fuente de la aplicación *LoadDB* desarrollada para el sistema diseñado como sustituto a la PBX del I.I.E. (por más información consultar el Capítulo 7).

Nombre de archivo	Directorio
`app_loaddb.c`	`\diseno_sistema_IIE\src\`

Requerimientos mínimos del sistema para visualizar el contenido del CD:

- Pentium II 300 MHz, 64 Mb de RAM, CD-ROM 12x y tarjeta de video SVGA.
- Sistema operativo Windows 98, ME, 2000 o XP.
- Adobe Acrobat Reader 4.0 o superior.
- WinRAR 3.11 o superior.

ACERCA DE ESTA DOCUMENTACIÓN

En esta sección se busca presentar la estructura de esta documentación, indicando el contenido de cada uno de los capítulos que la componen.

El **Capítulo 1 - Introducción** busca describir el marco en el que se encuadra el proyecto. Para ello se indican las principales motivaciones y objetivos del mismo.

El **Capítulo 2 - Conceptos Preliminares** introduce algunos elementos básicos, que son imprescindibles para la comprensión de la temática del proyecto. En primer lugar se describen los elementos fundamentales de arquitecturas VoIP. Se presentan aquellos aspectos que caracterizan la transmisión de voz sobre redes de conmutación de paquetes y los protocolos que posibilitan las aplicaciones de tiempo real (RTP y RTCP). Dejando el plano de datos y haciendo énfasis en el plano de control, se describen los protocolos de señalización VoIP más comúnmente utilizados, SIP y H.323, y se hace una comparación entre ambos que justifica la elección del primero para profundizar en nuestro estudio. Finalmente se hace una presentación de Asterisk, la PBX Open Source utilizada como plataforma de implementación de soluciones de telefonía.

El **Capítulo 3 - Elementos Fundamentales de Asterisk** intenta dar una visión más profunda de la PBX mediante una recorrida por sus 4 elementos constitutivos principales: los canales, el *dialplan*, la interfaz de línea de comando (CLI) y las aplicaciones. Es globalmente aceptado por la comunidad de usuarios el hecho de que hay que comprender el funcionamiento de estos cuatro elementos para entender Asterisk.

El **Capítulo 4 - El Canal SIP de Asterisk** describe el módulo que implementa la funcionalidad SIP en la PBX. En primer lugar se presentan las funcionalidades de Asterisk como entidad SIP. Luego se hace una recorrida por el archivo de configuración relativo al canal describiendo las opciones generalmente utilizadas. Finalmente se hace un seguimiento de bajo nivel de cómo procesa el sistema el arribo de una llamada de un usuario SIP registrado en el servidor.

El **Capítulo 5 - SIP & Asterisk - Primeros Acercamientos y Escenarios de Prueba** introduce diversas pruebas realizadas para el análisis del funcionamiento del sistema. Describe los diferentes clientes SIP utilizados en las pruebas de comunicación, enumerando sus funciones, con guías de configuración para su correcta interacción con Asterisk. Luego se presenta un estudio detallado del funcionamiento de Asterisk en entornos NAT. Finalmente se analizan las transferencias de llamadas, como funcionalidad avanzada de PBX, comparando las recomendaciones de la IETF para implementarlas en SIP contra la implementación propia de Asterisk.

El **Capítulo 6 - Soporte para IPv6 en el Canal SIP de Asterisk** describe las modificaciones desarrolladas para implementar dicha extensión de funcionalidad al sistema. En primer lugar, se presentan los aspectos fundamentales de este nuevo protocolo en comparación con IPv4. Luego se hace énfasis en el proceso de desarrollo, analizando: los elementos a modificar según las recomendaciones de la IETF y la arquitectura particular de Asterisk; el *patch* existente tomado como punto de partida; las modificaciones realizadas; y finalmente las pruebas para verificar su correcto funcionamiento.

El **Capítulo 7 - Diseño de un Sistema Basado en Asterisk** describe las etapas para el desarrollo de un sistema de central telefónica basado en Asterisk. El sistema que se buscó sustituir es el del Instituto de Ingeniería Eléctrica, para lo cual se debió hacer un estudio de los requerimientos del mismo, definir el hardware a comprar, y la configuración a nivel de Asterisk.

El **Capítulo 8 - Conclusiones** cierra esta documentación haciendo un balance final del proyecto SIP & *. Se sugieren algunos temas relacionados que pueden ser de interés para continuar con las investigaciones en el área de VoIP. Por último se hace un análisis en lo que respecta a la gestión del proyecto.

1. INTRODUCCIÓN

1.1. Motivación del Proyecto

El tráfico de voz sobre redes de paquetes, y especialmente Voz sobre IP (VoIP), está teniendo muy altos niveles de aceptación alrededor del mundo. Muchos analistas de la industria de las telecomunicaciones estiman que el mercado mundial de VoIP se tornará en un negocio multimillonario en pocos años. Por otro lado, el exponencial crecimiento que están teniendo las aplicaciones Open Source para sistemas operativos abiertos como Linux y sus respectivas comunidades de desarrolladores brindan un marco ideal para la evolución y el desarrollo de nuevas aplicaciones y servicios de telefonía sobre redes de paquetes.

"Me animo a predecir que en los próximos 3 años, VoIP, utilizando soluciones Open Source como Asterisk generará más negocios que el mercado entero de Linux hoy en día".

John "Maddog" Hall, Presidente de la Organización Linux International.
Octubre de 2004.

El hecho de que la mayoría de los protocolos y estándares utilizados para VoIP sean abiertos y estén disponibles públicamente motiva aún más a que instituciones académicas de investigación se enfoquen en estos temas para desarrollar sus propias aplicaciones. El Instituto de Ingeniería Eléctrica (I.I.E.) de la Facultad de Ingeniería, a través de su Grupo de Redes de Datos, ha estado impulsando gran parte de su investigación hacia los temas relativos a calidad de servicio (QoS) y performance de redes donde la transmisión de voz paquetizada es una de las aplicaciones de mayor interés. Existía entonces un interés en el I.I.E. por profundizar en los aspectos relativos a la implementación de VoIP. Una plataforma de pruebas como Asterisk, la PBX Open Source desarrollada íntegramente en software, parecía ideal teniendo en cuenta las restricciones presupuestales que limitarían las posibilidades de trabajar sobre equipamiento propietario. Más aún, para el I.I.E. tiene valor generar conocimiento acerca del funcionamiento y capacidades de Asterisk como solución telefónica de bajo costo, apuntando a posibles futuros convenios en esta área.

Este estudio, en su globalidad, permite disponer de una alternativa a las opciones de telefonía convencionales, lo cual resulta una opción sumamente razonable dada la creciente convergencia entre las redes de datos y de voz hacia redes multiservicio. A su vez, el hecho de usar software libre y desarrollar extensiones y mejoras al mismo permite extender de alguna forma el alcance de los resultados obtenidos en el marco de este trabajo, pues los mismos tendrán una mayor difusión.

Finalmente, el estudio de SIP como protocolo de señalización de llamadas VoIP se justifica plenamente por la gran importancia que ha tomado en los últimos años, absorbiendo gran parte del mercado dominado aún hoy por H.323. Su simplicidad y el empuje de aquellos *pro-Internet* lo han hecho pisar fuerte, aunque en redes con NAT presenta algunos problemas que pueden limitar su aplicación. Aún no existe una solución a este último problema que sea efectiva en todos los casos, pero tal vez se resuelva de manera indirecta con el advenimiento de IPv6.

1.2. Objetivos del Proyecto

El objetivo general del proyecto es el de implementar y testear estructuras de VoIP sobre una LAN, estudiando profundamente el protocolo de señalización SIP.

Podemos desagregar este objetivo en tres grandes bloques:

- Estudiar y comprender los aspectos fundamentales de VoIP: protocolos de señalización más comúnmente utilizados para la realización de llamadas telefónicas sobre redes IP, aspectos de calidad de servicio (QoS) relevantes y el equipamiento típico utilizado para brindar estos servicios. En particular, profundizar el estudio de SIP, enfocándose en sus ventajas y desventajas sobre el resto. El cumplimiento de este punto es importante para generar una base de conocimiento y encarar los siguientes objetivos.
- Estudiar las diferentes prestaciones de Asterisk manejando distintos protocolos de señalización VoIP (H.323, SIP, IAX) y obtener sólidos conocimientos de configuración y administración para realizar llamadas de aceptable calidad. Como integración de todos estos aspectos, realizar alguna práctica de diseño e implementación de un sistema de telefonía basado en Asterisk, sobre un caso específico con requerimientos reales en una aplicación de pequeño a mediano porte.
- Continuando el estudio del canal SIP de Asterisk, desarrollar mejoras al software para contemplar alguna funcionalidad requerida por aplicaciones de telefonía, que aún no fuera soportada por la PBX. Esto convertirá a Asterisk en una aplicación más potente de la cual el I.I.E. podrá hacer uso en proyectos futuros.

Siendo un proyecto de estudio de nuevas tecnologías con Asterisk como plataforma concreta que las implementa, surge la necesidad de dejar una base de conocimiento para futuros proyectos o aplicaciones en el área. Es por esto que se plantea como objetivo más allá de los requerimientos para la aprobación de la asignatura, crear una documentación clara, práctica y ejemplificada cubriendo:

- arquitectura de Asterisk y sus elementos constitutivos fundamentales.
- configuración y administración del sistema en diferentes escenarios de interés.
- descripción de las pruebas y mejoras realizadas con la respectiva evaluación de los resultados obtenidos.

En consecuencia, como criterio de aceptación del proyecto se evaluará si se ha adquirido un manejo fluido de Asterisk a nivel de administración y desarrollo del sistema. Dentro de una LAN, se deberán poder implementar comunicaciones VoIP integrando diversas tecnologías con las facilidades que brinda la PBX. Finalmente se deberá demostrar un buen dominio del protocolo SIP: sus fundamentos, ventajas y debilidades, y con más importancia aún, entender en detalle cómo se implementan las llamadas SIP en Asterisk.

2. CONCEPTOS PRELIMINARES

En este capítulo se presentan de manera introductoria y más bien informativa, los elementos en los que se basa el proyecto SIP & *.

En primer lugar se describen los elementos fundamentales de arquitecturas VoIP. Se presentan aquellos aspectos que caracterizan la transmisión de voz sobre redes de conmutación de paquetes y los protocolos que posibilitan las aplicaciones de tiempo real (RTP y RTCP).

Dejando el plano de datos y haciendo énfasis en el plano de control, se describe el protocolo de señalización para sesiones multimedia SIP, haciendo énfasis en el uso de esta tecnología en conjunto con *firewalls* y NAT. Dado que hoy por hoy el uso de estos elementos está ampliamente difundido, resulta interesante lograr un correcto funcionamiento de ambos con la tecnología SIP como exponente de VoIP.

También se describe el protocolo H.323 tradicionalmente utilizado para telefonía IP, haciendo luego una comparación entre éste y SIP dejando en claro las razones por las que se optó por profundizar en este último.

Finalmente se hace una introducción a Asterisk, PBX totalmente desarrollada en software Open Source, que utilizaremos como plataforma para el estudio de escenarios VoIP explorando sus funcionalidades y desarrollando algunas mejoras al sistema.

2.1. Voz sobre IP

2.1.1. Introducción

Tradicionalmente los tráficos de voz y de datos han sido soportados por redes distintas, diseñadas específicamente para proporcionar un servicio concreto. El primer tipo de redes, las de transporte de voz (telefonía) transportan un tráfico continuo en el tiempo. Se utilizan redes de conmutación de circuitos que asignan recursos a cada comunicación individual durante todo el tiempo para proporcionar un retardo constante. En el caso en el que no haya circuitos disponibles, la red rechaza la petición de un nuevo usuario.

El segundo tipo de redes, las de datos, proporciona servicio a un tráfico de naturaleza completamente distinta. Es un flujo de información intermitente por lo que la asignación de circuitos permanentes a lo largo de un periodo de tiempo supondría un desperdicio de recursos en

los intervalos de inactividad. Por este motivo las redes de datos son redes de conmutación de paquetes. En este caso la asignación de recursos es dinámica y sólo se consumen recursos cuando existe tráfico que transmitir. El retardo es variable debido a las fluctuaciones en el nivel de ocupación de las colas de paquetes.

Al utilizar dos redes independientes para el flujo de datos y de voz es posible optimizarlas por separado. Sin embargo, al integrar múltiples servicios sobre una red se consigue una reducción de costos y una utilización más eficiente de los recursos.

El primer intento de integración fue la RDSI (Red Digital de Servicios Integrados) que supone la transmisión de datos sobre redes de voz. El usuario accede a servicios en modo paquete o en modo circuito a través de una interfaz que lo conecta a la red. El segundo enfoque es *Frame Relay*, que es el intento en sentido contrario: utilizar una red de datos para transmitir voz. El tercer intento es ATM (Modo de Transferencia Asíncrono), que transporta datos de forma eficiente pero permite asegurar calidad de servicio para la voz. En este caso, la integración de servicios se produce en toda la red y no en la interfaz como ocurre en RDSI.

En los últimos años el éxito de Internet ha llevado a tratar de integrar todos los servicios sobre una red basada en IP. Aunque el motivo inicial era la reducción en los costos de usuarios y operadores, a medida que se fueron implementando nuevas soluciones se evidenciaron los múltiples servicios que se pueden ofrecer (conferencias multimedia, mensajería unificada - voz, correo electrónico y fax -, servicios de teleoperador a través de la web, indicadores de presencia, etc.).

Sin embargo para poder ofrecer servicios sobre IP es necesario que la red pueda proporcionar calidad adecuada a cada tipo de servicio. Este aspecto es crítico en los servicios multimedia (voz y video) donde las exigencias de retardo son cruciales. Al utilizar redes IP para transmisión de voz es necesario solucionar numerosos problemas: pérdida de paquetes, retardos grandes y muy variables, sobrecarga debido a las cabeceras de los paquetes de voz, etc. Estos temas relativos a QoS han adquirido mucha importancia y motivado el desarrollo de diversas arquitecturas para minimizar los efectos nocivos intrínsecos a las redes IP.

Inicialmente, la IETF ha propuesto dos arquitecturas para ofrecer QoS en IP: IntServ (Servicios Integrados) y DiffServ (Servicios Diferenciados). Estos mecanismos no han sido suficientes, por lo cual se ha comenzado a trabajar en aspectos de Ingeniería de Tráfico. Su principal objeto es mapear la demanda de tráfico sobre la topología de la red y poder controlar el tráfico de la misma. MPLS (*Multiprotocol Label Switching*, RFC[1] 3031) ha surgido como un protocolo que permite realizar Ingeniería de Tráfico ya que posibilita el ruteo explícito (desde el origen), pudiéndose asignar diferentes caminos a las diferentes clases de tráfico. Por más información

[1] RFC (*Request for Comments*) es documentación estándar sobre Internet. Incluye estándares, recomendaciones y documentos informativos. Pueden ser publicados por cualquiera, aunque la IETF es una de sus principales fuentes. La IETF (*Internet Engineering Task Force*) es una comunidad internacional de diseñadores de red, operadores, vendedores e investigadores involucrados en la evolución de la estructura y el modo de operación de Internet.

sobre los aspectos de QoS en redes IP consultar [5].

En el caso del servicio de telefonía, además de solucionar los problemas técnicos derivados de la transmisión de voz sobre paquetes IP, es necesario cumplir con requisitos de calidad de voz, disponibilidad del servicio, conectividad entre terminales de diferentes redes, seguridad; es decir, el servicio de telefonía que se transmite a través de redes IP debe cumplir los requisitos exigidos a la telefonía convencional. Para alcanzar estos requisitos se ha definido una arquitectura del plano de control, separada del plano de datos por los nuevos controladores de llamadas, y protocolos de señalización entre los planos. En lo que sigue, se hará una breve introducción a los conceptos relativos a la transmisión de voz paquetizada.

2.1.2. Transporte de voz sobre paquetes

En esta sección se presentan algunos de los problemas asociados al transporte de voz sobre paquetes que aparecen en las diferentes tecnologías de paquetes (VoIP, VoATM, VoFR). La figura 2.1 ilustra las funciones básicas necesarias para el transporte de voz sobre una red de conmutación de paquetes.

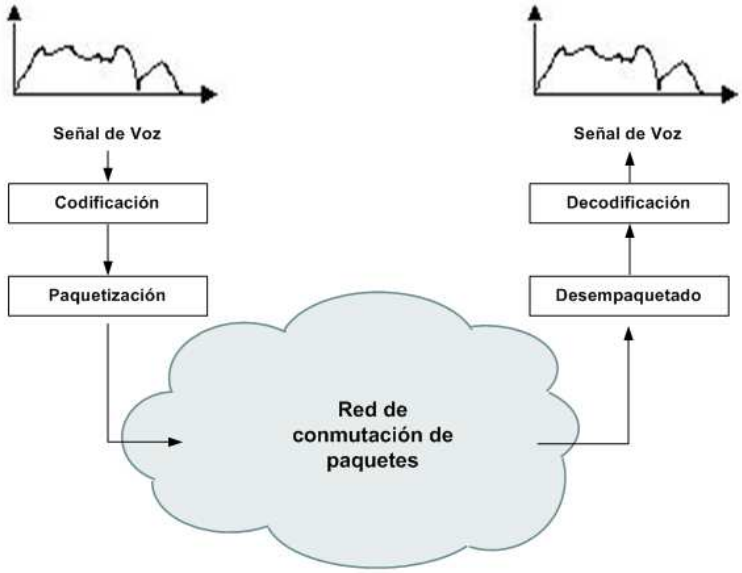


Figura 2.1: Transporte de voz sobre red IP.

El primer paso del proceso es el muestreo y digitalización de la señal vocal en el origen. A continuación se codifica la información en bloques obtenidos con una ventana temporal (el número de bits por bloque depende del codificador). El tercer paso es la paquetización, en la que se encapsulan uno o más bloques de datos en un paquete (la paquetización se produce en uno o varios pasos, dependiendo de la pila de protocolos que se utilice).

El proceso en el extremo de recepción es inverso al realizado en el origen: desempaquetado, decodificación y reconstrucción de la señal a partir de la señal digitalizada.

En apariencia es un proceso sencillo pero está condicionado por los problemas derivados del transporte del flujo de información a través de la red. Según la carga de tráfico de los diferentes nodos, los paquetes sufren esperas variables dando lugar a fluctuaciones en el retardo (*jitter*) y en algunos casos a pérdidas de paquetes. Esto da lugar a un flujo de paquetes que no mantiene el espaciamiento constante y que es posiblemente incompleto. Desde el punto de vista del usuario estos problemas se traducen en la degradación de la voz.

En la definición de redes ATM se tuvo en cuenta a la voz como servicio importante y se aseguraron diferentes grados de calidad de servicio. Sin embargo las redes IP y *Frame Relay*, pensadas originalmente para datos, no incluyen requisitos estrictos de retardo y es necesario incluir una serie de mecanismos para obtener calidad de servicio adecuada para voz.

Los mecanismos complementarios que reducen los efectos indeseados (pérdida de paquetes y fluctuaciones de retardo) se listan a continuación:

- Codecs compresores que reducen el ancho de banda necesario.
- Buffers en recepción que regeneran el espaciado entre paquetes y amortiguan las fluctuaciones del retardo.
- Mecanismos de calidad de servicio en la red (por ejemplo, priorización de los paquetes de voz).

2.1.3. Codecs de voz

Existen tres tipos de codificadores de voz, según las técnicas de codificación que emplean:

De forma de onda: representan cada muestra del conversor analógico-digital con un valor binario. Un ejemplo representativo es el G.711 utilizado en la red telefónica conmutada.

Adaptativo diferencial: es muy parecido al codec de forma de onda, pero no codifica el valor de la muestra sino la diferencia entre cada muestra y la anterior. La ventaja fundamental que presenta es que necesita *bit rates* menores porque el rango de valores que tiene que codificar es mucho más reducido.

Predictivo lineal (LPC): son codificadores de bloques que trabajan con conjuntos de muestras en vez de utilizar muestras aisladas. Seleccionan el conjunto a través de ventanas de muestreo, extraen una serie de características y las representan de manera parametrizada. Una de sus principales virtudes es la baja tasa binaria que generan. Al comparar la tasa binaria con la del G.711 se observan reducciones significativas por lo que se denominan codecs compresores. Estos codificadores se utilizan en ocasiones en las que el ancho de banda es un recurso escaso. Un ejemplo es el acceso vía módem telefónico a la telefonía IP a través de Internet. Otro caso es el de un operador que ofrece tránsito de llamadas de larga distancia sobre una red IP.

En la tabla 2.1 se presentan las características más relevantes de los principales codecs de voz normalizados en recomendaciones de la serie G del ITU-T y el codec utilizado en GSM "*Full Rate*" (GSM 06.10).

	G.711	**G.721**	**G.726**	**G.728**	**G.729**	**G.723.1**	**GSM FR**
Tipo de codificación	PCM	ADPCM	ADPCM	LD-CELP	CS-ACELP	MP-MLQ/ ACELP	RPE-LTP
***Bit rate* (kbps)**	64	32	16/24/ 32/40	16	8	6.4/5.3	13
Complejidad (MIPS)	0.1	10	12	33	22	16/8	2.5
Retardo de codificación (ms)	0.125	0.125	0.625	0.125	15	37.5	20
Calidad (MOS)	4.2	4.0	4.0	4.0	4.0	3.7-3.9	3.6-3.8

Tabla 2.1: Codecs normalizados.

Las principales características que conviene destacar son:

- El *factor de compresión*, indica la reducción de ancho de banda que proporciona. Se compara con los 64 kbps habituales en la PSTN.
- La *complejidad del algoritmo de codificación*, es directamente proporcional a la capacidad de proceso necesaria (influye en aspectos de implementación: si puede implementarse en software o requiere hardware específico). Normalmente el algoritmo es más complejo al aumentar el factor de compresión del codec.
- *Retardo de codificación*, depende del tipo de proceso que se realiza sobre las muestras. En los codecs LPC es superior a los demás porque es necesario esperar a obtener el grupo de muestras antes de codificar. El retardo puede ser despreciable en comparación con el resto, pero si existen problemas de retardo es necesario optar por un codificador más sencillo.
- *Calidad*, evaluada mediante el parámetro MOS (*Mean Opinion Score*) que se obtiene a partir de la valoración subjetiva de un conjunto de personas. Se están proponiendo nuevos modelos (*E-model* de ETSI[2]) porque es un método que no tiene en cuenta las pérdidas o los errores.

2.1.4. Supresión de silencios

Es un mecanismo complementario al empleo de codecs compresores para reducir el ancho de banda. Se pretende detectar períodos de silencio durante la conversación (mecanismos VAD, *Voice Activity Detection*) suprimiendo el envío de paquetes de voz mientras dure la situación.

[2] ETSI es el Instituto Europeo de Normas de Telecomunicación. El *E-model* es un modelo matemático que incluye parámetros como S/N, interferencias simultáneas a la señal de voz, interferencias retrasadas, etc.

Como en la conversación telefónica cada interlocutor sólo habla la mitad del tiempo y realiza pausas entre frases, se pueden obtener reducciones de hasta el 60 % en el flujo de paquetes.

Para evitar que el interlocutor piense que se ha cortado la comunicación durante los intervalos de silencio se envían periódicamente paquetes de silencio (SID, *Silence Insertion Description*) durante la pausa. Estos paquetes proporcionan una indicación del nivel de ruido que existe en el origen para que el receptor lo simule en el terminal remoto mediante un generador de ruido (Recomendación ITU-T I.366.2).

2.1.5. Tamaño de los paquetes de voz

La elección del tamaño de los paquetes de voz es otro de los factores críticos, porque los encabezados que se añaden a los paquetes generan un tráfico añadido a la tasa normal del codec. Si se desea minimizar el impacto de los encabezados en el tráfico, es preciso enviarlos el menor número de veces posible y para ello se envían paquetes de gran tamaño con varios bloques de datos en cada paquete. Esta solución presenta un gran inconveniente porque cuanto mayor es el número de bloques de voz por paquete, mayor es el tiempo de paquetización (tiempo que hay que esperar para llenar el paquete).

El retardo extremo a extremo es uno de los aspectos más críticos de los sistemas de voz paquetizada y será lo que determine el tamaño máximo de paquete posible.

Como la compresión de voz aumenta el tiempo de paquetización, cuanto más complejo sea el algoritmo de compresión, menores serán los paquetes.

2.1.6. Retardo extremo a extremo

Existen muchos factores que contribuyen al retardo extremo a extremo: retardo del algoritmo de codificación, tiempo de paquetización, tiempo de propagación (despreciable salvo en distancias muy grandes), tiempo de transmisión, tiempos de espera en los nodos de conmutación (dependiente del tráfico en la red), tiempo de descompresión, etc.

El retardo total extremo a extremo en una conversación telefónica, ha de mantenerse por debajo de un cierto nivel para minimizar dos efectos indeseables: la pérdida de interactividad y el eco.

A partir de 150 ms de retardo en un sentido, la comunicación se vuelve molesta por la pérdida de interactividad: la persona que habla, al percibir que su interlocutor tarda en contestar, repite sus palabras, a la vez que recibe la respuesta procedente del otro extremo.

El segundo efecto, el eco, es más molesto cuanto mayor es el retardo ida y vuelta. Los motivos por los que se produce eco en la telefonía IP son los mismos de las redes telefónicas convencionales. Sin embargo, el retardo que introduce el transporte de voz por redes de paquetes (codificación, paquetización, transporte y espera en los nodos) hace que los efectos sean más

perjudiciales.

Eco eléctrico: el bucle telefónico convencional consta de un par de hilos sobre los que se transmite de manera bidireccional. En el teléfono y en las centrales telefónicas se separan los dos sentidos de transmisión mediante bobinas híbridas. Como estos dispositivos no son perfectos, la separación de señales no es completa y aparecen reflejos indeseados de las señales hacia los focos emisores. De todas las posibles reflexiones, la más molesta es la que presenta mayor desfasaje temporal con respecto a la señal original. Si el retardo ida y vuelta de la señal es elevado (superior a 50 ms) el usuario percibe el eco. En la telefonía convencional este retardo solamente se produce en redes telefónicas muy grandes. Sin embargo, en los *gateways* VoIP este límite se supera con bastante frecuencia. Al superar 50 ms de retardo, es necesario utilizar mecanismos de supresión o cancelación de eco.

Eco acústico: es el que se produce por acople entre el altavoz y el micrófono. Suele ser despreciable en terminales telefónicos convencionales y sin embargo tiene entidad suficiente en equipos manos libres o en teléfonos móviles que incluyen sus propios mecanismos de cancelación. En un entorno de telefonía IP con PC, que integran altavoz y micrófono en el equipo, es necesario tener en cuenta el eco acústico.

En la teoría se plantea que el punto óptimo de cancelación de eco es el punto más próximo a la reflexión. Sin embargo, en la práctica hay que tener en cuenta que el eco ha de ser cancelado por el operador que lo introduce.

2.1.7. Fluctuación de retardo (*jitter*)

El transporte de voz paquetizada no es sensible sólo al retardo extremo a extremo (latencia) sino también a las fluctuaciones o variaciones de ese tiempo de retardo (*jitter*). Estas variaciones son debidas a la fluctuación en los tiempos de espera de los nodos de conmutación de la red que dependen del tráfico concreto del momento.

Estas variaciones en el tiempo de retardo se traducen en flujos de paquetes espaciados de manera irregular en el tiempo. Como la regeneración de la señal de voz en el receptor es un proceso síncrono, necesita disponer de un bloque de voz de manera periódica (con periodo dependiente de la ventana de muestreo del codificador). Sin embargo, por las variaciones del retardo, el flujo de paquetes recibido carece de la sincronía necesaria. Para evitarlo se utiliza un buffer amortiguador en el receptor que almacena los paquetes por orden de llegada extrayéndolos de manera síncrona.

El tamaño del buffer refleja el compromiso entre el filtrado de las variaciones de retardo (mejor cuanto mayor sea el buffer) y el retardo extra que se añade (menor cuanto menor sea el buffer).

En el caso de voz sobre ATM la red puede ofrecer conexiones con retardo máximo (Rmax) y con variación máxima (Rmax-Rmin) garantizados por lo que el empleo de un buffer amortiguador solucionaría el problema de *jitter*.

En las redes IP actuales no hay garantía de retardo máximo por lo que existe una cierta probabilidad de pérdida de paquetes. En este caso es necesario ajustar el buffer en función de las pérdidas admisibles y el retardo adicional tolerable. En la práctica muchos equipos de VoIP suelen ajustar el buffer de manera automática en función de las características reales del tráfico.

2.1.8. Pérdida de paquetes

Las pérdidas de paquetes se producen por errores de transmisión y por congestión de la red. Como el tráfico es impredecible en las redes de paquetes no orientadas a conexión, el número de paquetes que esperan a salir por un enlace en un momento determinado puede superar la capacidad de la cola de salida. En este caso, el conmutador debe descartar los paquetes que no caben en la cola y estos se pierden. En algunos casos se descartan paquetes antes de que se produzca la congestión para que los emisores de flujo perciban las pérdidas como aviso de congestión inminente y reduzcan el tráfico.

En los emisores de voz, que tienen flujo de tasa fija, el aviso de congestión es inútil porque no pueden reducir la tasa de envío aunque se detecten las pérdidas. En este caso tampoco se pueden aplicar técnicas de retransmisión porque los paquetes serían descartados en el destino por llegar con retardo excesivo. Por este motivo en el caso de VoIP se utiliza el protocolo de transporte UDP en lugar de TCP.

Para solucionar estos problemas, las redes de paquetes emplean diferentes mecanismos:

- ATM tiene mecanismos normalizados de control de tráfico y calidad de servicio.
- *Frame Relay* no garantiza calidad de servicio pero puede controlar el tráfico entrante y evitar situaciones de congestión.
- Las redes IP no tienen ningún mecanismo y sólo pueden ofrecer un servicio *best-effort*. El problema de la calidad de servicio IP sigue abierto, pero han habido importantes avances con MPLS como herramienta para realizar ingeniería de tráfico.

Además se emplean técnicas de corrección de errores, reducción de variaciones de retardo, enmascaramiento[3] de los paquetes de voz perdidos y si no queda otro remedio, sobredimensionar la red (solución más típica actualmente en VoIP).

[3] Para evitar que el usuario escuche sonidos extraños al perder un paquete, una posibilidad es introducir un sonido menos molesto mientras dura la interrupción, como por ejemplo ruido blanco (solución empleada en GSM). En otros casos se utilizan técnicas de interpolación reemplazando el paquete perdido por uno obtenido a partir del anterior y del posterior.

2.1.9. Protocolos de "tiempo real" sobre IP

Los protocolos de tiempo real para la transmisión de audio y video por Internet se definen dentro de la RFC 1889. Esta norma incluye dos protocolos que constituyen el estándar de facto: RTP y RTCP. El primero, protocolo RTP, regula el intercambio de información en diferentes formatos (audio y video). RTCP regula la comunicación de control que se establece entre los extremos, en paralelo con la transmisión de información. Su objetivo es proporcionar información actual del estado y la calidad de la comunicación.

La norma RFC 1889 no establece qué protocolos deben utilizarse en las capas inferiores, por debajo de RTP/RTCP. Sin embargo, en la mayor parte de los casos se emplea UDP.

En la figura 2.2 se presenta la arquitectura de protocolos empleada en el intercambio de voz o video entre dos terminales VoIP conectados a través de la red IP.

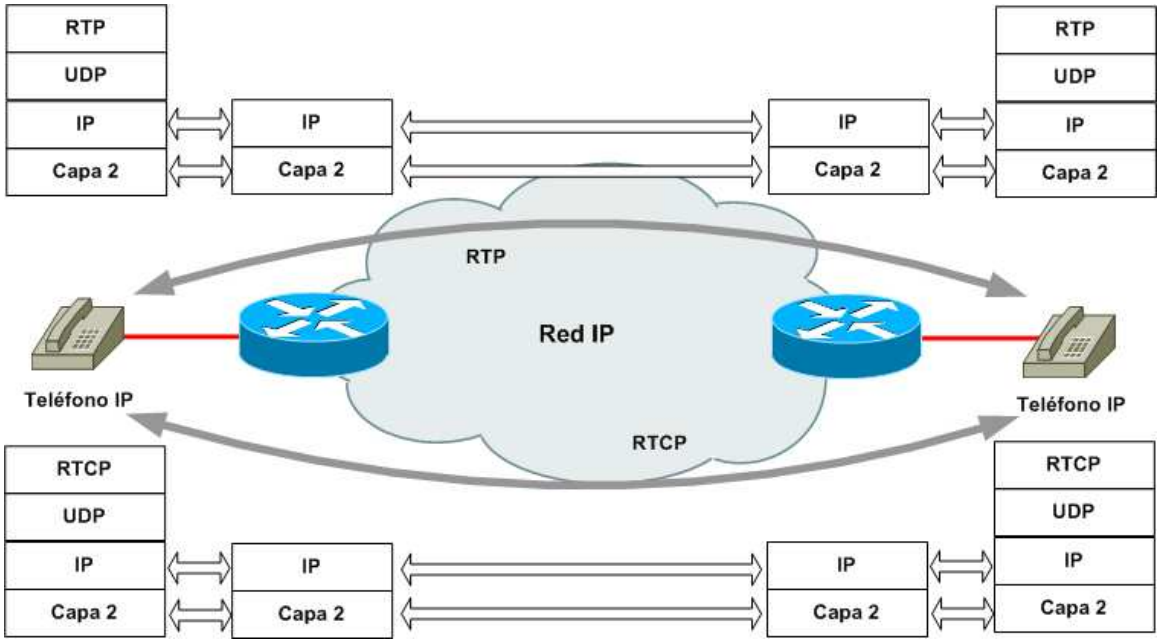


Figura 2.2: Tráfico RTP y RTCP a través de la red.

El flujo de paquetes RTP (en los que se incluyen los bloques de voz o video) se transportan mediante paquetes UDP. Para el intercambio de paquetes RTP, la norma establece los puertos UDP pares, elegidos de manera independiente en cada extremo de la comunicación. Para notificar al extremo remoto el puerto local seleccionado se utiliza un mecanismo de señalización que queda fuera del ámbito de la RFC 1889 (por ejemplo H.323 o SIP, que se verán más adelante).

Como se observa en la figura 2.2, el flujo de paquetes de control RTCP es paralelo al flujo de

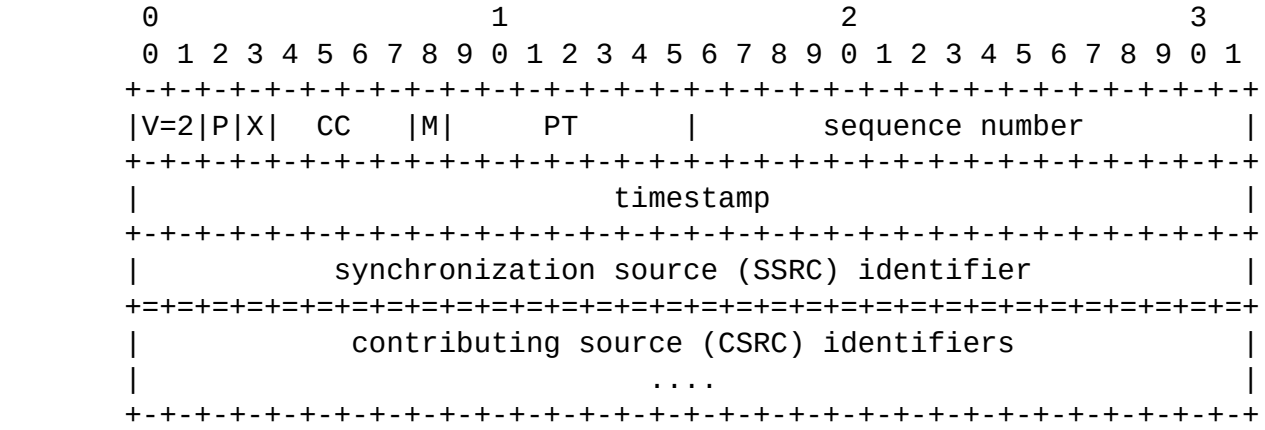
paquetes RTP. Los paquetes RTCP se intercambian de manera periódica. En el caso en el que se emplee UDP para el tráfico RTCP, se utilizan los puertos UDP inmediatamente superiores a los empleados en el flujo RTP.

RTP

El protocolo RTP (*Real Time Protocol*) es el estándar para el transporte de tráfico (audio o video) en tiempo real sobre Internet. Su origen se encuentra en la red MBONE (*Multicast Backbone*) que es una red virtual de difusión (multicast), superpuesta sobre Internet, empleada en el desarrollo de multiconferencias a través de Internet. En el protocolo RTP se asume la existencia de imperfecciones en la red (pérdidas y retardos) y la posibilidad de variación de las características de la red durante la comunicación.

Formato de la trama RTP

Los paquetes RTP transportan la información (bloques de audio o tramas de video) y un encabezado que permite la reconstrucción del flujo de datos en el receptor. Esta estructura se representa a continuación.



Los campos de encabezado que aparecen en esta trama son:

V (versión, 2 bits): versión del protocolo RTP (la versión actual definida en la RFC 1889 es la 2).

P (relleno, 1 bit): P=1 indica que se ha incluido relleno al contenido del paquete (para algunos algoritmos de cifrado que utilizan bloques de tamaño fijo).

X (extensión, 1 bit): indica que se han incluido campos de encabezado adicionales (para intercambiar más información de control).

CC (cuenta de CSRC, 4 bits): número de identificadores CSRC presentes en el encabezado. Útil cuando hay un mezclador porque permite especificar el número de flujos que han sido mezclados.

M (marca, 1 bit): la interpretación de este campo depende del campo PT (tipo de carga útil). Por ejemplo, puede servir para indicar el final de un cuadro de video o el inicio de una ráfaga de voz tras un intervalo de silencio.

PT (tipo de carga, 7 bits): formato de carga útil. Existen códigos normalizados que se han definido para distintos tipos de codecs de audio o video.

SN (número de secuencia, 16 bits): aumenta una unidad al enviar un paquete RTP. Permite la detección de pérdida de paquetes en destino.

Timestamp (instante de muestreo, 32 bits): representa el valor del reloj de muestreo para el primer bit de carga útil.

SSRC (identificador de fuente, 32 bits): identifica la fuente que envía el paquete RTP.

CSRC (identificador de contribuidores, de 0 a 15 identificadores de 32 bits cada uno): identificadores de las fuentes que envían la mezcla incluida en la carga útil. Hay tantos identificadores como se indique en el campo CC.

Ancho de banda para una sesión VoIP

Al estimar el ancho de banda necesario para una sesión VoIP es necesario tener en cuenta el tamaño de carga útil y la sobrecarga por encabezados. El tamaño de carga útil viene determinado por el tamaño de los bloques de información que entrega el codificador y por el número de bloques que se desean transportar en un paquete. En el cálculo de la sobrecarga por encabezados se tienen en cuenta los encabezados que añaden los sucesivos protocolos (RTP, UDP, IP y capas inferiores). En el caso más sencillo, por ejemplo en una sesión de voz entre dos terminales VoIP, el encabezado RTP se compone de 12 octetos, a los que hay que sumar los 8 del encabezado UDP y los 20 de IP. Los octetos de los niveles inferiores dependen de la tecnología concreta utilizada (por ejemplo 6 octetos para PPP[4]).

En la tabla 2.2 se presenta el ancho de banda necesario para una llamada de voz en dos casos diferentes: codec normalizado G.711 de 64 kbps (codec de la red telefónica) y el G.729 de 8 kbps (codec de baja velocidad utilizado en el acceso a una red IP vía módem).

En los valores de la tabla 2.2 se observa claramente la influencia del tamaño total de la carga útil en la tasa total de envío y en el retardo de paquetización. Cuanto mayor es la carga útil, menor es la tasa de envío (se reduce la sobrecarga por encabezados) pero mayor es el retardo (aumenta el tiempo de construcción del paquete).

Los valores de la tabla 2.2 indican que, en algunos casos, el transporte de voz sobre IP puede ser muy ineficiente. Para reducir el ancho de banda necesario se puede recurrir a diferentes métodos que se presentan en el apartado siguiente.

[4] PPP es el estándar de Internet para transmisión de paquetes IP sobre líneas serie.

Codec	Tasa nominal kbps	Retardo paquetización (ms)	Tamaño carga útil octetos	*Bit Rate* (kbps)		
				IP (sin capa 2)	IP/PPP	IP/AAL5
G.711	64	5	40	128	137.6	169.6
		10	80	96	100.8	127.2
		20	160	80	82.4	106
G.729	8	10	10	40	44.8	84.8
		20	20	24	26.4	42.4
		40	40	16	17.2	21.2

Tabla 2.2: Tasa de envío nominal y real en dos codecs normalizados.

Reducción del ancho de banda en una sesión VoIP

Los métodos más habituales para reducir el ancho de banda son la supresión de silencios y la compresión de encabezados.

Los mecanismos de compresión de encabezados se aplican en el enlace, es decir que si un extremo utiliza este método, el extremo remoto es responsable de la descompresión. Un ejemplo típico es el enlace de acceso vía módem desde un PC hasta el servidor de acceso remoto de un ISP (Proveedor de Servicio de Internet).

Existen varios estándares de compresión de encabezados (RFC 2508 - CRTP - y RFC 3095 - ROCH -) que permiten la compresión de encabezados IP, UDP y RTP. Estos mecanismos suprimen la información redundante que se transmite repetidamente en una sesión RTP y consiguen reducir la sobrecarga de encabezados a 2 o 4 octetos. Esta reducción supone una mejora sustancial de la eficiencia. Sin embargo, la compresión de encabezados sólo suele aplicarse en enlaces de acceso y no en enlaces troncales, en los que se recurre a la supresión de silencios.

RTCP

RTCP (*Real Time Control Protocol*) regula el intercambio de mensajes de control entre los participantes en una sesión multimedia. Esta información se refiere, fundamentalmente, a la calidad de servicio con que se está desarrollando la comunicación: retardo, *jitter*, tasa de paquetes recibidos y perdidos, etc. Es una comunicación paralela a la transmisión de información, que se establece entre los extremos, de forma opcional. Aunque es opcional, es recomendable porque proporciona el estado actual de la comunicación (determina si la calidad de la transmisión es suficiente) y permite tomar las medidas oportunas durante la transmisión de información (por ejemplo utilizar un codec con menor tasa, o adaptar el tamaño del buffer en recepción). Es un protocolo para informar sobre la calidad del servicio pero no puede mejorar

las prestaciones de la red, es decir no proporciona mecanismos de calidad de servicio[5].

Funciones de RTCP

Además de información de calidad de servicio, RTCP proporciona otras funciones adicionales que resultan de gran utilidad en escenarios con múltiples participantes:

Identificación: intercambio de identificadores entre participantes (nombre, e-mail, número de teléfono, etc.).

Correlación de relojes: permite medir el retardo extremo a extremo de los paquetes RTP al proporcionar la correlación entre el reloj local (muestreo de las fuentes) y el tiempo global.

Control: notificaciones de control de los participantes (abandono de un participante o intercambio de notas de texto entre participantes).

Mensajes de RTCP

La RFC 1889 define cinco tipos de mensajes RTCP:

SR (Sender Report): estadísticas de transmisión y recepción de los participantes activos (los que generan y reciben tráfico).

RR (Receiver Report): estadísticas de recepción de los participantes pasivos (sólo reciben).

SDES (Source Description): mensajes que permiten asociar identificadores de fuente con los datos del usuario: nombre, login, e-mail, etc. y opcionalmente: ubicación del usuario, número de teléfono o aplicación que utiliza.

BYE: fin de participación (en escenarios con más de dos participantes no supone el fin de la sesión).

APP (Application-Specific): intercambio de información entre las aplicaciones que están utilizando RTP.

Ancho de banda de RTCP

El uso de RTCP consume un ancho de banda añadido al RTP. Supone entre 1 y 5 % del ancho de banda de RTP. Aunque no es un tráfico excesivo, en sesiones con múltiples participantes es necesario controlarlo para evitar avalanchas.

[5] RTCP no proporciona mecanismos como reserva de ancho de banda, control de congestión o prioridades de tráfico; sólo proporciona información útil para evitar en lo posible las imperfecciones de la red.

2.2. SIP - Session Initiation Protocol

El Protocolo de Inicio de Sesión (SIP, *Session Initiation Protocol*), es un estándar de la IETF que se especifica en la RFC 3261 [3].

Como se explica en un extracto de dicha RFC: "... el Protocolo de Inicio de Sesión (SIP) es un protocolo (de señalización) de control de aplicación de capas para crear, modificar y cerrar sesiones con uno o más participantes. Estas sesiones incluyen conferencias multimedia, llamadas telefónicas, distribución multimedia por Internet, etc.".

SIP es uno de los protocolos que forman la arquitectura IETF para una comunicación multimedia escalable, en tiempo real y multiparte. Algunos de los protocolos más destacados de esta arquitectura (excluyendo aquellos relacionados con QoS) son:

- RTP y RTCP, que se especifican en la RFC 1889, proporcionan una entrega de medios en tiempo real.
- El Protocolo de Flujo en Tiempo Real (RTSP, *Real-Time Streaming Protocol*), que se especifica en la RFC 2326, proporciona una entrega bajo demanda de datos en tiempo real.
- El Protocolo de Descripción de Sesión (SDP, *Session Description Protocol*), que se especifica en la RFC 2327, proporciona un formato de descripción estándar para el intercambio de capacidad de medios (como los codecs de voz para VoIP).

En la figura 2.3 se muestran los protocolos que definen la arquitectura IETF para comunicaciones multimedia. En ella se puede ver la interacción entre el plano de datos y el plano de control, y el fundamental rol que juega SDP como vínculo entre ambos. Notar que las flechas azules representan vínculos lógicos, y no transmisión real de datos a través de la red.

En cuanto a las solicitudes SIP, éstas permiten establecer sesiones y enviar descripciones de las mismas. SIP soporta sesiones unicast y multicast, así como llamadas punto a punto y punto-multipunto.

La extensibilidad de SIP permite el desarrollo de nuevas funcionalidades para la telefonía sobre IP. Los encabezados de SIP son versátiles, y se pueden registrar funcionalidades adicionales a través de la Agencia de Asignación de Números de Internet (IANA, *Internet Assigned Numbers Authority*). La flexibilidad mencionada también permite construir servicios de telefonía avanzados, como aquellos que incluyen movilidad. Todo esto se discutirá más profundamente en la siguiente sección.

2.2.1. Atributos de SIP

SIP se diseñó como solución a largo plazo para las conferencias multimedia y la telefonía en Internet. Se han tenido muchas consideraciones con el desarrollo del protocolo para asegurarse de que es una plataforma viable para futuras comunicaciones que se basen en Internet:

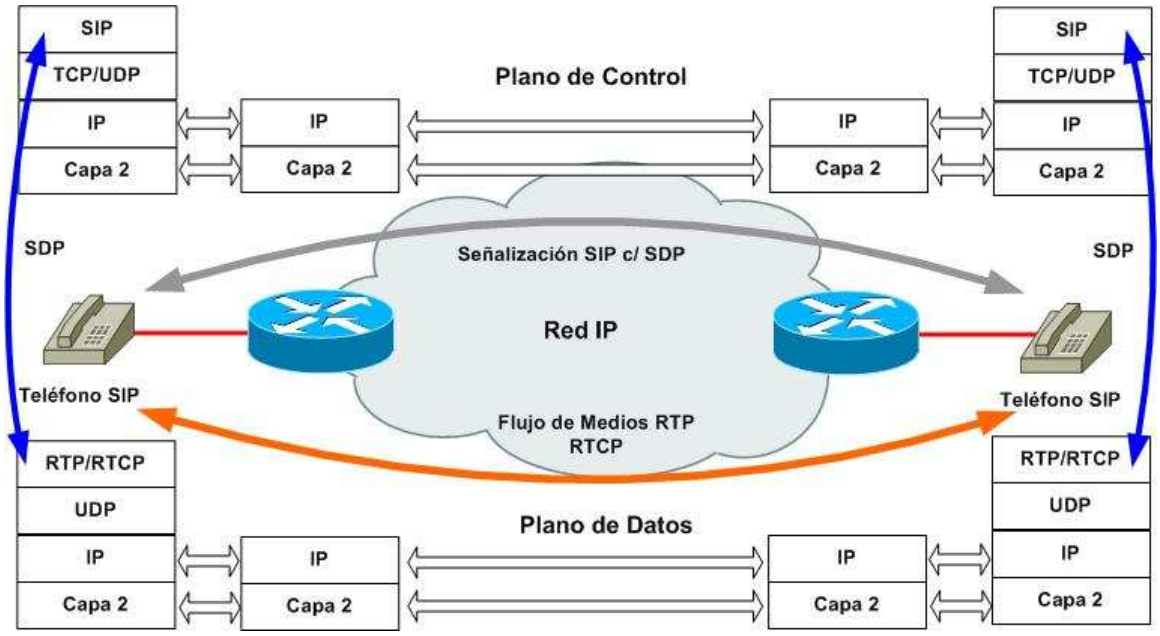


Figura 2.3: Arquitectura SIP.

Simplicidad: a diferencia de la mayoría de los protocolos de telefonía en Internet (como, sin ir más lejos, H.323), SIP emplea mensajes en texto plano. Esto hace que el protocolo sea relativamente sencillo para resolver problemas e integrarse con otras aplicaciones.

Eficiencia: el protocolo superior de SIP tiene poco impacto en la eficiencia de la comunicación, ya que las funciones de señalización consumen poco ancho de banda en relación con el flujo de medios. SIP es muy eficaz en términos de tiempo de conexión de llamada, ya que toda la información necesaria para el establecimiento de la llamada se incluye en el mensaje inicial.

Escalabilidad: los servidores no necesitan mantener la información del estado de las sesiones basadas en los mensajes SIP que procesan, por lo que un solo servidor puede manipular eficientemente muchos clientes. A su vez, SIP detecta y previene los bucles de enrutamiento de mensajes, que pueden consumir grandes recursos de la red y son más comunes a medida que ésta crece.

Flexibilidad: dado que SIP usa SDP para negociar los codecs, se puede utilizar cualquier codec registrado por la IANA. Aquellos codecs no registrados deben compartir un campo genérico para los "codecs no estándar".

Soporte de movilidad: el modelo de comunicación SIP se dirige a los usuarios, que se pueden mover de terminal a terminal (por ejemplo, teléfonos y computadoras). El protocolo proporciona un importante soporte para *proxying* y redirección, por lo que los usuarios tienen la opción de proporcionar u ocultar su verdadera ubicación.

Extensión: la arquitectura de SIP es modular y flexible, lo cual favorece su crecimiento y el agregado de extensiones, mientras asegura una operación más cercana a las versiones más antiguas. También facilita la posibilidad de retirar las opciones no usadas, evitando que el protocolo se convierta en poco manejable.

2.2.2. Funcionalidad básica y elementos claves de SIP

Los dos componentes fundamentales de un sistema SIP son los *agentes de usuario* (UA), y los *servidores de red*. Tanto el usuario llamante como el llamado son identificados por medio de *direcciones SIP* (llamadas "URIs SIP").

Agentes de usuario

Los agentes de usuario (de aquí en más UA) son aplicaciones que inician o terminan una comunicación SIP. Son capaces de enviar y recibir solicitudes SIP, y respuestas a esas solicitudes SIP.

Contienen tanto un Cliente de Agente de Usuario (UAC) como un Servidor de Agente de Usuario (UAS), también conocidos como cliente y servidor respectivamente. UAC y UAS son entidades lógicas. UAC es la parte del UA que envía solicitudes y recibe respuestas. UAS es, por el contrario, la parte del UA que recibe solicitudes y envía respuestas.

Se observa que todos los mensajes SIP se clasifican en una de esas categorías: o son solicitudes, o son respuestas. Esta es una característica muy importante del protocolo.

Servidores de red

Hay tres tipos de servidores de red: los servidores *proxy*, los servidores de redirección y los servidores de registro (registradores):

Los *registradores* son entidades SIP especiales, que aceptan registros de clientes que indican las direcciones en las que se los puede localizar. Una vez que el registrador extrae la ubicación del usuario a partir del registro que éste envió, la almacena en una base de datos que luego pueden consultar los servidores *proxy* o de *redirección*.

Los *servidores proxy* son entidades muy importantes en la infraestructura SIP. Llevan a cabo el enrutamiento de las solicitudes que forman parte de una sesión, de acuerdo a la ubicación actual del usuario llamado (destino). Luego, un servidor *proxy* recibe las solicitudes de los UASs que se encuentren en la red, y las reenvía de modo de que éstas lleguen a destino.

La información que emplea un servidor *proxy* para dirigir solicitudes SIP va más allá del ámbito de la especificación SIP. Este tipo de servidores puede tener conocimiento de la ubicación de los agentes de usuario locales, ya sea a partir de la información de un registrador

SIP, o por algún otro método de búsqueda.

Por otro lado, un *servidor de redirección* es una entidad que acepta solicitudes SIP, pero no las procesa, sino que envía a modo de respuesta una lista conteniendo todas las posibles ubicaciones de los usuarios destino de dichas solicitudes. Para conocer esas ubicaciones, el servidor de redirección consulta la base de datos creada por un registrador.

2.2.3. Direccionamiento

Aunque los servidores *proxy*, los servidores de redirección y los registradores pueden participar en una transacción SIP, sólo los usuarios y los agentes de usuario tienen direcciones SIP. Los servidores SIP se identifican por las direcciones IP y los puertos TCP/UDP. Por defecto, estos servidores escuchan en los puertos TCP y UDP 5060, pero pueden utilizar cualquier otro.

Sintaxis de dirección SIP

Las direcciones SIP, también llamadas SIP Universal Resource Locators (URLs), tienen el formato `usuario@servidor` (muy similares a las direcciones de e-mail). El nombre que se le da a este tipo de direcciones es "Uniform Resource Identifiers (URI): Generic Syntax". Un URI SIP básico tiene, en su parte de usuario, un nombre o un número telefónico, mientras que la porción de servidor puede ser el nombre de un dominio o una dirección de red. En general, su forma es:

sip:usuario:clave@servidor:puerto;parámetros-uri?encabezados

Un URI SIP puede tener varios parámetros y encabezados. Algunos ejemplos de URI SIPs se muestran en el cuadro 2.3. Donde aparece, el símbolo `%20` es el valor codificado del caracter de espacio.

```
sip:pepe@compania.com;transport=udp
sip:pepe:clave_pepe@compania.com;transport=udp;user=ip
sip:pepe@192.168.1.1:5060
sip:pepe@compania.com:5060;transport=tcp?subject=test%20TCP%20call
```

Tabla 2.3: Ejemplos de SIP URIs.

En cuanto al parámetro `user`, si se tiene `user=phone` entonces se permite el soporte para direcciones E.164. De esta forma, mediante el uso de este parámetro, SIP permite distinguir los puntos finales E.164 (correspondientes a números telefónicos) de los puntos finales IP regulares. Esto permite usar en el campo de usuario un número telefónico convencional.

2.2.4. Ubicación del servidor

Para los clientes SIP es común utilizar un servidor *proxy* local como el próximo salto para todas las solicitudes SIP salientes. En consecuencia, los servidores *proxy* deben saber cómo ubicar a los distintos agentes de usuario para poder enrutar correctamente dichas solicitudes.

Generalmente, las direcciones SIP contienen nombres de dominio DNS que indican cuál es la ubicación del usuario llamado; un ejemplo de esto sería la URI SIP `sip:pepe@compania.com`. En este caso, al recibir una solicitud con esta URI SIP como destino, el *proxy* deberá ser capaz de identificar a qué dirección de red corresponde el nombre `compania.com`, para lo cual recurrirá a un servidor DNS. Pero para que todo esto funcione correctamente, el cliente debe saber, en primer lugar, cómo ubicar al servidor *proxy* en cuestión.

Por otro lado, los servidores *proxy* también pueden utilizar la información almacenada en un registrador SIP, que les indicará dónde encontrar al cliente objeto de la consulta. Pero para tener la información actualizada, el registrador debe recibir los registros de los usuarios, y para eso es necesario que los mismos sepan cómo encontrarlo.

Cómo encontrar un servidor proxy o de redirección

La especificación SIP básica no proporciona un método para que los clientes UA aprendan automáticamente acerca de un *proxy* local o de los servidores de redirección.

Un método posible consiste en que el cliente UA ubique al servidor por medio de una consulta DNS, que devuelve la dirección IP del servidor. La ventaja de esto es que se pueden añadir o eliminar los servidores SIP que sean necesarios sin la necesidad de actualizar cada cliente.

Otra opción que existe actualmente consiste en que el cliente UA se encuentre periódicamente descargando un archivo de configuración que contenga el nombre de dominio DNS, o la dirección IP y el puerto TCP/UDP del servidor SIP. Esta opción es muy práctica ya que permite que los clientes SIP sean completamente portátiles.

Cómo encontrar un registrador

Los agentes de usuario SIP se pueden configurar estáticamente para la localización de un registrador SIP, o pueden emplear la multidifusión para encontrarlo. Hay también un nombre DNS para estas direcciones.

Por su parte, los restantes agentes de usuario SIP de la red pueden escuchar también las direcciones multidifusión del registrador SIP para detectar la presencia de otros agentes de usuario. Esta técnica puede ser un mecanismo eficiente para que los agentes de usuario desarrollen mallas localizables en un dominio administrativo, cuando no existe un servidor *proxy* local. Sin embargo, puede suponer un trabajo intensivo para el procesador de los clientes,

cuando hay un gran número de ellos en el dominio multidifusión. Es por esto que esta solución no es implementada por ningún cliente.

2.2.5. Transacciones SIP

Luego de resuelto el direccionamiento, el cliente envía una o más solicitudes SIP y recibe una o más respuestas desde el servidor especificado. Todas estas solicitudes y respuestas se consideran parte de una transacción SIP.

Las transacciones SIP pueden hacerse sobre UDP o TCP. En el caso de TCP, se pueden enviar todos los mensajes de solicitud y respuesta relativos a una única transacción SIP sobre la misma conexión TCP. También se pueden llevar transacciones SIP diferentes entre dos entidades sobre una misma conexión TCP. Si se usa UDP, la respuesta se envía a la dirección identificada en el encabezado del pedido.

2.2.6. Mensajes SIP

Existen tres partes en un mensaje SIP: la "línea de inicio", los encabezados y el cuerpo.

La "línea de inicio" indica el propósito del mensaje, los encabezados proporcionan los detalles del mismo y el cuerpo opcional suministra detalles añadidos que no encajan en los encabezados. Las tres secciones siguientes explorarán estas partes del mensaje.

"Línea de inicio"

Mediante el término "línea de inicio" se hace referencia a la primera línea del mensaje SIP. El formato de la misma depende de si el mensaje es una respuesta o una solicitud (se recuerda que todos los mensajes SIP entran en alguna de estas categorías). Luego, será posible distinguir qué tipo de mensaje SIP se está procesando tan sólo a partir de su "línea de inicio".

A continuación se describirá la forma de la "línea de inicio" para las solicitudes SIP, y para las respuestas SIP.

Solicitudes SIP

El formato de la "línea de inicio" para las **solicitudes SIP** es el siguiente:

`<Método> <Solicitud-URI> <Versión SIP>`

Donde `<Método>` es uno de los tipos de solicitudes SIP (que se describen a continuación); `<Solicitud-URI>` es el URI SIP de la entidad que debería recibir el mensaje; y `<Versión SIP>` es actualmente SIP/2.0.

Hay seis tipos de solicitudes SIP, también llamadas métodos SIP. Estos son:

INVITE Inicia una llamada VoIP. Normalmente invita a la/s parte/s llamada/s a unirse a la sesión. Lleva el mensaje SDP que contiene sus capacidades para el intercambio de medios.

ACK Reconoce el haber recibido una respuesta final a un INVITE. El establecimiento de una sesión usa *handshake* de tres vías, debido a la naturaleza asimétrica de la invitación. El usuario llamado retransmite periódicamente una respuesta final positiva hasta que recibe un ACK (lo cual indica que el llamante aún está del otro lado listo para comunicarse).

OPTIONS Habilita a consultar y adquirir las capacidades de los servidores UA. La respuesta del servidor debe incluir los métodos SIP que soporta.

BYE Es usado tanto por el usuario llamado como por el llamante para manifestar su deseo de liberar una llamada.

CANCEL Cancela una solicitud pendiente. Es usado por ejemplo cuando un servidor *proxy* cancela búsquedas UAS en paralelo, después de una respuesta con éxito, o cuando el usuario llamado no ha respondido todavía pero el llamante quiere abortar el intento de comunicación.

REGISTER Es usado por los clientes para registrar información de su ubicación en los registradores.

Respuestas SIP

Por otro lado, el formato de la "línea de inicio" para las **respuestas SIP** es el siguiente:

`<Versión SIP> <Código de respuesta> <Frase razón>`

Donde `<Versión SIP>` es actualmente SIP/2.0, `<Código de respuesta>` es un código de identificación de la respuesta, y `<Frase razón>` da una descripción de la misma.

Los servidores contestan a las solicitudes de un cliente con una o más respuestas. Las respuestas SIP estándar están codificadas con tres dígitos de código de respuesta y una descripción textual que la sigue.

Se clasifican en seis categorías, identificadas por el primer dígito del código de respuesta:

1xx - Provisional: una respuesta provisional es aquella que le indica a su receptor que la solicitud asociada ha sido recibida pero que el resultado de su procesamiento aún no es conocido. Este tipo de respuestas son enviadas únicamente cuando el procesamiento no es inmediato. El que envió la solicitud debe dejar de reenviarla una vez que recibe este tipo de respuesta. Típicamente, los servidores *proxy* envían respuestas con código 100 (*Trying* - Intentando) cuando comienzan a procesar un INVITE, y los agentes de usuario envían respuestas con código 180 (*Ringing* - Llamando) lo cual implica que el teléfono del usuario llamado está timbrando.

2xx - Éxito: la acción solicitada fue recibida, comprendida y aceptada. Típicamente se trata del código 200 *OK*.

3xx - Redirección: el cliente debe realizar una acción adicional para completar la solicitud. Este tipo de respuesta brinda información sobre la nueva ubicación del usuario o un servicio alternativo que puede usar el usuario llamante para llevar a cabo su llamada. Normalmente este tipo de respuesta es enviada por los servidores de redirección. La ubicación que lleva la respuesta puede ser o bien la correspondiente a otro servidor *proxy* o bien la nueva ubicación del usuario llamado (extraída de la base de datos del registrador). Este tipo de respuestas es final.

4xx - Fallo en la Solicitud: la solicitud no es válida en este servidor en concreto, pero puede ser válida en algún otro. Esto se puede deber a que o bien la sintaxis de la solicitud no es correcta, o bien a que el servidor en cuestión no la puede procesar.

5xx - Fallo en el Servidor: la solicitud puede ser válida, pero este servidor en concreto falló al procesarla, por otras razones. Los clientes usualmente deberían intentar enviar la solicitud de nuevo más tarde.

6xx - Fallo Global: la solicitud fallará siempre (no lo intente en otros servidores). Por ejemplo, este tipo de respuesta aparece cuando un usuario no quiere participar en una conversación.

SIP introduce la categoría de códigos 6xx para distinguir entre las respuestas de error que pudiesen producirse en cualquier servidor y las respuestas de error que son únicas para un servidor en concreto. Por ejemplo, un servidor UA puede declinar la solicitud de un cliente porque está ocupado (lo cual es una respuesta de error por parte de un servidor), pero el cliente necesita saber si deberá seguir con la solicitud en otra parte. El usuario llamado puede que desee estar inlocalizable para cualquier origen (no molestar en una configuración global), por lo que el UAS puede informar al cliente de su estado con una respuesta 600 *Busy Everywhere*. El cliente entonces sabe que tiene que dejar de enviar solicitudes INVITE a otros servidores UA del usuario.

Si un cliente recibe un código de respuesta SIP no reconocido, negocia la respuesta como una respuesta x00 para una categoría determinada. Este comportamiento asegura la compatibilidad hacia atrás con versiones anteriores de SIP, además de permitir extensiones para los códigos de respuesta SIP.

Encabezados

Los encabezados especifican el usuario llamante, el usuario llamado, la ruta y el tipo de mensaje.

Cabe mencionar que, por simplicidad y consistencia, los encabezados de todos los mensajes de solicitud y respuesta correspondientes a una misma transacción SIP, son los mismos.

Los cuatro grupos de encabezados son:

Encabezados genéricos: se aplican a solicitudes y respuestas. Son por ejemplo: `Accept`, `Call-ID`, `To`, `From`, `Expires`.

Encabezados de entidad: definen información sobre el tipo de cuerpo del mensaje y su largo. Son `Content-Encoding`, `Content-Length` y `Content-Type`.

Encabezados de solicitud: permiten al cliente incluir información adicional de solicitud. Son por ejemplo: `Authorization`, `Contact`, `Route`.

Encabezados de respuesta: permiten al cliente incluir información adicional de respuesta. Son por ejemplo: `Allow`, `Proxy-Authenticate`, `Warning`.

2.2.7. Ejemplo de llamada SIP

En la figura 2.4 se muestra un ejemplo típico de intercambio de mensajes SIP para el establecimiento de una llamada entre Alicia y Benjamín. Supongamos que Alicia utiliza un *softphone* corriendo en su notebook (`rosaluna.fing.edu.uy`) para llamar vía Internet a Benjamín que tiene un teléfono SIP conectado en su LAN. Se muestran dos servidores *proxy* locales a las redes de cada uno, que se encargarán del establecimiento de la sesión. Ésta es una configuración clásica conocida como el *trapecio SIP*.

Alicia llama a Benjamín utilizando su URI SIP, en este caso `benjamin@mit.edu`. Supongamos que Alicia posee un URI `alicia@fing.edu.uy` ya que se encuentra en otro dominio, `fing.edu.uy`. La primera transacción de este ejemplo se inicia cuando el *softphone* de Alicia envía un mensaje de INVITE con destino al URI SIP de Benjamín. Dentro de la información que se envía en el mensaje INVITE podemos destacar un identificador único de la llamada, dirección de destino y origen, e información relativa a la sesión que Alicia planea iniciar (mensaje SDP). Dicho mensaje INVITE tendrá la siguiente forma:

```
INVITE sip:benjamin@mit.edu SIP/2.0
Via:SIP/2.0/UDP rosaluna.fing.edu.uy;branch=z9hG4bK776asdhds
Max-Forwards: 70
To: Benjamin <sip:benjamin@mit.edu>
From: Alicia <sip:alicia@fing.edu.uy>;tag=1928301774
Call-ID:a84b4c76e66710@rosaluna.fing.edu.uy
CSeq:314159 INVITE
Contact:<sip:alicia@rosaluna.fing.edu.uy>
Content-Type:application/sdp Content-Length: 142
```

Como el *softphone* no conoce la ubicación ni de Benjamín ni del servidor SIP en el dominio `mit.edu`, el INVITE se envía al servidor *proxy* SIP local a la red de Alicia. En general la dirección de este servidor debió haberse configurado en su softphone. El servidor *proxy* en `fing.edu.uy` recibe la solicitud de Alicia y devuelve una respuesta 100 TRYING, indicando que el mensaje anterior fue recibido y que se está encargando de enrutar el INVITE al destino. El servidor *proxy* en `fing.edu.uy` localiza al servidor en `mit.edu` por algún mecanismo,

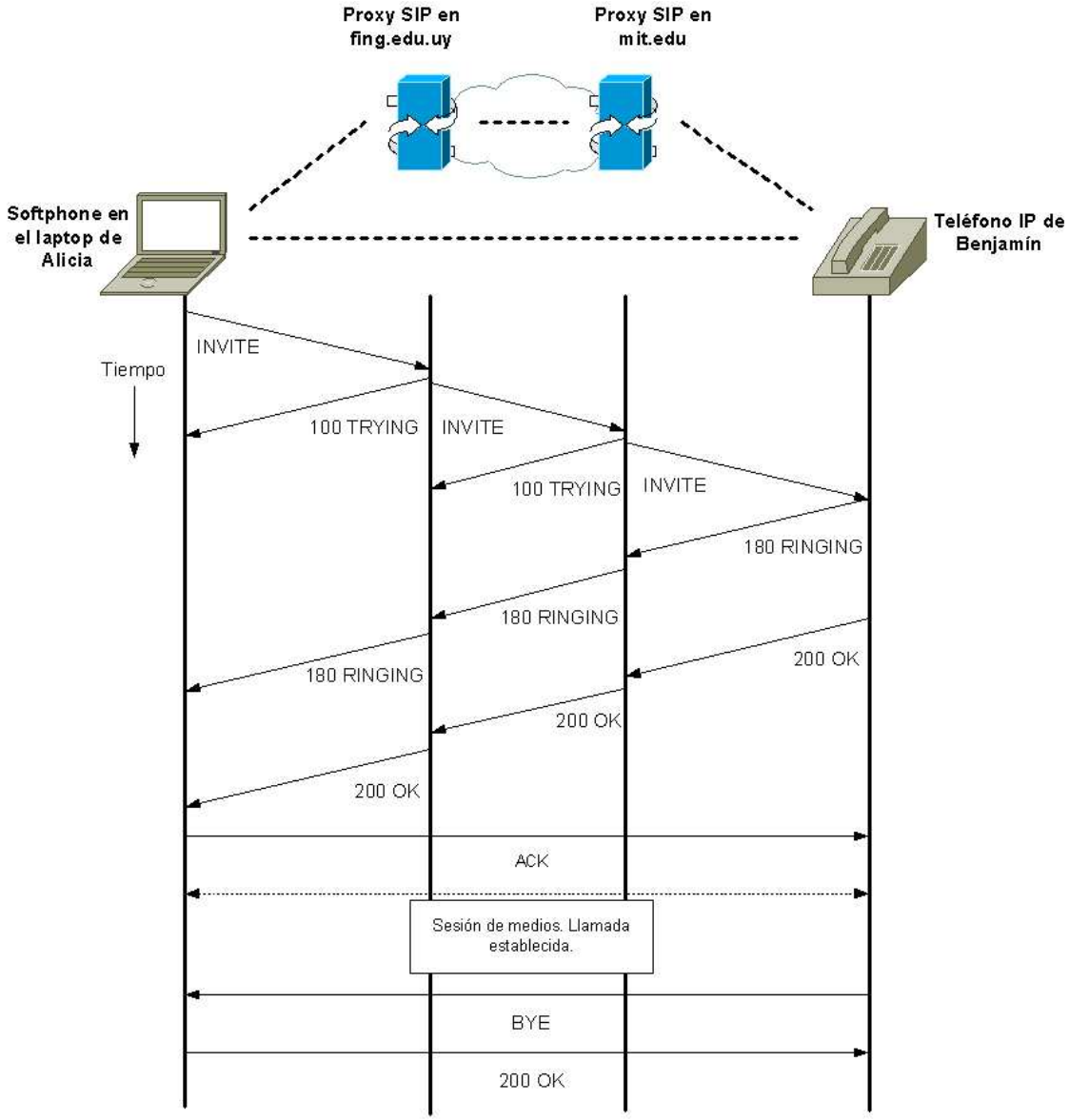


Figura 2.4: Intercambio de mensajes en llamada SIP.

posiblemente una consulta DNS especial. Luego, una vez que conoce la dirección IP de este último servidor, le reenvía la solicitud INVITE de Alicia. El servidor *proxy* consultará su base de datos de usuarios registrados en búsqueda de la dirección actual de Benjamín, para finalmente enviarle el mensaje INVITE.

El teléfono de Benjamín comenzará a sonar y lo indica enviando respuestas 180 RINGING, que se enrutan de vuelta hasta Alicia a través de los dos *proxies*. Es importante notar que en este punto cada uno de los *proxies* utiliza el encabezado `Via` para determinar la dirección de

origen a la cual enviar las respuestas y no se necesitan consultas DNS adicionales. Además se tiene la interesante propiedad de que cada *proxy* que maneje un INVITE, observará todas las respuestas al mismo.

Cuando el *softphone* de Alicia recibe la respuesta 180 RINGING, generará tonos audibles de indicación y algún mensaje en pantalla para indicar que el teléfono de Benjamín está sonando. Como Benjamín decide atender, su teléfono envía una respuesta 200 OK para indicar este hecho y también incluye la información sobre el tipo de sesión que desea establecer (mensaje SDP). La forma de este mensaje de respuesta se muestra a continuación:

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP asterisk10.mit.edu
   ;branch=z9hG4bKnashds8;received=192.0.2.3
Via: SIP/2.0/UDP asterisk.fing.edu.uy
   ;branch=z9hG4bK77ef4c2312983.1;received=192.0.2.2
Via: SIP/2.0/UDP rosaluna.fing.edu.uy
   ;branch=z9hG4bK776asdhds ;received=192.0.2.1
To: Benjamin <sip:benjamin@mit.edu>;tag=a6c85cf
From: Alicia <sip:alicia@fing.edu.uy>;tag=1928301774
Call-ID: a84b4c76e66710@rosaluna.fing.edu.uy
CSeq:314159 INVITE
Contact: <sip:benjamin@192.0.2.4>
Content-Type: application/sdp
Content-Length: 131
```

Cabe mencionar que, en su ruta, los sucesivos servidores *proxy* que atraviesa el mensaje van quitando sus correspondientes encabezados `Via`.

Cuando este mensaje llega al *softphone* de Alicia, se establece la comunicación y se envía un mensaje ACK. Éste se enruta directamente a Benjamín sin pasar por los *proxies*, ya que a esta altura ambos terminales conocen sus direcciones mediante la información del encabezado `Contact` en los mensajes INVITE/200 OK ya intercambiados.

La sesión de medios entre Alicia y Benjamín ha comenzado y pueden hablar enviando paquetes de acuerdo a los codecs y demás requerimientos acordados mediante el intercambio SDP. Notar que en general los paquetes de voz van por un camino diferente al que toman los mensajes de señalización SIP.

Para terminar la llamada Benjamín cuelga su microteléfono y comienza una nueva transacción enviando un mensaje BYE. El *softphone* de Alicia responde con un mensaje 200 OK y termina la llamada. En este caso, no se envía un ACK, ya que éste sólo se utiliza para confirmar una solicitud INVITE.

2.2.8. Firewalls, NAT y SIP

Los *firewalls* y el NAT *(Network Address Translation)* afectan a los protocolos de señalización de telefonía IP, dificultando la tarea de llamar a terminales que se encuentran fuera de la red

privada o protegida. A pesar de que por lo general *firewalls* y NAT van de la mano, estos representan dos problemas bien diferentes que serán presentados en esta sección.

Firewalls y telefonía IP

Tanto las llamadas SIP como las llamadas H.323 usan una cierta cantidad de puertos distintos, de los cuales sólo los de señalización están bien definidos: el puerto 1720 de TCP para H.323, y el puerto 5060 de TCP/UDP para SIP.

Una vez que comenzó la señalización, se necesitan nuevos canales. H.323 usualmente utiliza una conexión TCP nueva para el intercambio de capacidades (mediante H.245), lo cual hace uso de puertos dinámicamente asignados. De igual forma, el flujo de medios por RTP usa puertos dinámicamente asignados en cada lado. La única restricción que se aplica es que estos puertos deben ser superiores al 1023.

Debido a esto, para poder dar telefonía IP a una zona protegida por un *firewall*, se debe elegir una de las siguientes opciones:

- El *firewall* debe dejar abiertos los puertos por arriba del 1023.
- Los clientes deben tener configurado (y usar) un puerto RTP determinado, y ese puerto debe habilitarse en el *firewall*.
- El *firewall* debe ser "consciente" de que se está desarrollando telefonía IP, es decir, debe monitorear todos los mensajes SIP y H.323 de forma de abrir y cerrar los puertos requeridos según sea necesario, *on the fly*.
- Se puede hacer uso de un *proxy* SIP o H.323 fuera de la zona protegida por el *firewall*, configurando a éste último para permitir comunicaciones punto a punto sólo a través del *proxy*. Ésta es una solución de mediana seguridad, ya que permite una comunicación relativamente segura entre puntos protegidos y un servidor *proxy* confiable fuera del *firewall*.

NAT y telefonía IP

Un problema diferente es el que surge cuando parte de la zona de telefonía IP reside dentro de una red privada, donde no hay direcciones IP públicas. En este caso pueden generarse las siguientes complicaciones:

Registro: considérese un punto final que reside dentro de una red privada, y se registra en un servidor público. Sin saber de la existencia del NAT, podría intentar registrarse con su IP privada. Por ende, eventualmente la respuesta del servidor podría llegar a destino, pero esto no será así para ninguna llamada que el servidor intente hacerle llegar más adelante.

Señalización de llamadas: algunos mensajes de señalización de llamadas contienen direcciones IP necesarias para proseguir la comunicación (por ejemplo, la dirección IP y puerto a los que los datos multimedia deberán ser enviados). Usualmente el usuario final transmite su dirección IP local (privada), haciendo que el usuario que mantiene la comunicación con él falle al intentar conectarse a esa IP.

Una posible solución a esto es el protocolo STUN, definido en la RFC 3489. Un usuario final implementando este protocolo se conecta a un servidor STUN público para conocer su dirección IP pública. Luego, el resultado de esta consulta es usado como dirección IP para registrarse con un servidor remoto. Cabe mencionar que esta solución es muy popular en el mundo de SIP, no así en el de H.323.

Otra posibilidad consiste en usar un *router* que sea "consciente" de los protocolos de telefonía IP y re-escriba las direcciones IP que se encuentran dentro de los mensajes a nivel de capa de aplicación, cuando estos son enrutados. Sin embargo, esto requiere un soporte para decodificación y codificación completa de los mensajes SIP y H.323, lo cual es simple en el caso de SIP (dado que es un protocolo basado en texto) pero complejo en el caso de H.323 (pues es basado en ASN.1). Además, siempre es posible que un *router* de estas características borre el contenido de algunos mensajes que no conoce, por lo que esta solución fallaría si se quisieran transportar nuevos aspectos de un protocolo. Un *router* de estas características puede tener la forma de una aplicación corriendo en el propio *router* NAT.

SIP y NAT

Atravesar NAT es uno de los temas más problemáticos de SIP. Para comenzar a entender en qué consiste el problema: en una Internet ideal, todos los dispositivos serían capaces de comunicarse punta a punta sin ningún intermediario (excepto los *routers*). Esto asume que cada dispositivo tiene una dirección IP alcanzable, es decir una identidad de Internet pública, alcanzable y única.

En la realidad, hoy muchos de los dispositivos conectados a Internet se encuentran detrás de un NAT. Si bien esto impide que desde Internet se inicien conexiones al dispositivo (lo cual es malo para la telefonía IP u otras formas de comunicaciones *peer-to-peer*), por este método el usuario es protegido contra ataques maliciosos.

Por otro lado, usando NAT uno también puede conectar múltiples dispositivos a Internet usando sólo una dirección IP pública. En resumen, se puede decir que NAT tiene sus ventajas, y también sus desventajas.

¿Por qué SIP no trabaja correctamente cuando está detrás de NAT? La razón es que muchos de los parámetros de comunicación de SIP son transmitidos dentro del propio mensaje SIP; entre esos parámetros se encuentran la dirección IP y los números de puerto usados para la señalización y el intercambio de medios. Un dispositivo SIP detrás de NAT no sabe cómo será "visto" desde Internet, sólo conoce su propia dirección IP y los puertos donde corre la

aplicación SIP.

Una vez que la comunicación con Internet comienza, el dispositivo NAT traduce la combinación `dirección IP privada:puerto` del dispositivo SIP conectado a la interfaz privada del NAT, a un mapeo temporario de `dirección IP pública:puerto` en la interfaz conectada a la Internet.

¿Cómo puede resolverse el problema de atravesar NAT en ambas direcciones para el tráfico SIP? El tráfico SIP está formado por señalización y medios; usualmente viajan en caminos paralelos y usan distintos números de puerto. Por lo tanto, ambos necesitan ser tratados en formas diferentes.

Señalización SIP

Para que un dispositivo SIP con una dirección privada sea alcanzable, debe primero conectarse a Internet. Esto puede hacerse a través del método REGISTER de SIP, que guarda en un registrador SIP la ubicación actual del dispositivo. Cuando comienza la sesión, el registrador SIP es consultado para conocer la dirección actual de Internet del dispositivo.

STUN (*Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)* [16] es un estándar diseñado para ayudar a los dispositivos SIP a que se den cuenta de cómo son vistos desde Internet. El dispositivo SIP puede usar los valores que le envíe un servidor STUN (alcanzable desde la Internet pública), para la señalización SIP. Sin embargo, dependiendo del tipo de NAT, el dispositivo NAT puede hacer diferentes mapeos para cada nueva combinación de `dirección IP:puerto` que le llega. Esto hace que la información brindada por el servidor STUN no sirva, y no se podrán realizar llamadas SIP con ningún dispositivo de la Internet. En conclusión, STUN no logra proveer una solución 100 % segura para atravesar NAT.

El problema del NAT para la señalización SIP puede resolverse simplemente a través de la implementación de un registrador inteligente. Éste no debe almacenar la dirección de contacto tal como se la envía el dispositivo en el mensaje de registro (REGISTER), sino que debe guardar la combinación real de `dirección IP:puerto` de donde viene el mensaje.

Una vez registrado en el registrador SIP, tanto el dispositivo como el registrador deben mantener el canal de comunicación levantado enviando paquetes *keep-alive*. De esta forma, el mapeo hecho en el NAT no expira. Estos paquetes pueden ser tanto paquetes SIP enviados por el dispositivo, o paquetes IP enviados por el registrador.

Dependiendo del tipo de NAT, puede ocurrir que enviar paquetes *keep-alive* desde el exterior no sea suficiente; en este caso, el dispositivo dentro del NAT debería tener esto en cuenta y setear el período de envío de mensajes de registro por abajo del tiempo de expiración del mapeo del NAT (usualmente un valor de 85 segundos es suficiente).

Ahora, al tener un camino de comunicación permanentemente abierto entre el registrador y el dispositivo SIP, siempre es posible llamar al dispositivo detrás del NAT y comenzar a negociar una sesión SIP.

El único requisito, que afortunadamente ahora está disponible en la mayoría de los dispositivos SIP, es usar señalización simétrica, que implica que el dispositivo debe recibir y enviar datos en el mismo número de puerto.

Flujo de medios

El flujo de medios consiste en uno o múltiples flujos que son negociados en la señalización SIP. Los flujos de medios pueden ser incorporados o eliminados de la comunicación entre los dispositivos SIP. Como esto ocurre dinámicamente, uno debe ser capaz de traducir en tiempo real los mapeos entre las direcciones públicas y privadas.

A través del mensaje SIP INVITE inicial, los dispositivos SIP negocian sus capacidades de medios; esto se hace a través del protocolo SDP, que le permite a SIP obtener información sobre los flujos de medios (direcciones en las que estos serán recibidos, tipos de codecs a usar, ancho de banda, etc.). Como SDP contiene información sobre la dirección IP del dispositivo SIP, al estar presente el NAT, la IP que se informará será la privada, lo cual es un problema.

Existen dos formas de resolverlo. Una es "mejorar" los dispositivos SIP para que sean capaces de negociar dinámicamente un camino para el flujo de medios aún incluso luego de que la sesión de SIP haya sido establecida. Esto puede ser logrado a través de ICE (*Interactive Connection Establishment*), que le permite a los dispositivos probar múltiples caminos de comunicación a través del uso de distintos números de puerto y técnicas de STUN. Si ambos dispositivos tienen soporte ICE, es muy probable que se pueda establecer la comunicación de medios punta a punta sin ningún intermediario.

Dependiendo del tipo de NAT, puede ocurrir que aún teniendo soporte ICE en ambas puntas, no se pueda establecer la comunicación correctamente. En este caso, es necesario usar un *relay* de medios. El principal inconveniente de esto es que al tener que atravesar un nuevo punto, el camino entre las puntas es más largo lo cual puede provocar que el retardo punta a punta sea muy alto.

Para paliar el problema del retardo, el *relay* de medios debe intercalarse en algún punto del camino más corto entre los dos dispositivos finales. Esto puede ser logrado teniendo varios *relays* de medios conectados en las cercanías de los usuarios que son atendidos por un mismo proveedor SIP.

TURN (*Traversal Using Relay NAT* [17]) es una recomendación de la IETF, que implementa *relays* de medios para puntos finales SIP. El enfoque que hace del problema dista de ser ideal: asume que los clientes tienen una relación de confianza con un servidor TURN y que las sesiones se establecen usando credenciales compartidas.

Esto tiene un problema de escalabilidad, y requiere cambios complejos en los clientes SIP, ya que TURN es un protocolo difícil de implementar, que no tiene posibilidad de distribuir la carga y complica la configuración del UA SIP.

Otro enfoque al mismo problema, que no implica cambios en los clientes SIP, es aprovechar la relación de confianza que existe entre los dispositivos SIP y el *proxy* SIP. En contraste con la forma de trabajar de TURN, en este caso es el *proxy* SIP y no el UA SIP el que hace la reserva de sesión para el intercambio de medios. La primera ventaja de esto es que el UA SIP no tiene que contar con ninguna capacidad de TURN, y además no es necesario almacenar ninguna base de datos de credenciales en el servidor TURN y en el cliente.

Otra ventaja se desprende del hecho de que el *proxy* SIP siempre tiene más idea de dónde ubicar el *relay* de medios para una sesión, que los dispositivos SIP en sí. Esto permite la ubicación de un *relay* de medios por cada sesión establecida en un lugar óptimo de Internet, y resuelve el balance de carga y la escalabilidad de la función de retransmisión de medios.

Hasta aquí se analizó cómo puede resolverse el problema del NAT de forma óptima y escalable mediante la distribución de funciones entre el *proxy* SIP, los agentes de usuario y los *relays* de medios. Este enfoque se opone a la solución que ofrecen los *Session Border Controllers*, que representan un punto donde convergen la señalización SIP y los medios antes de salir a la red SIP. Al tener *Session Border Controllers* en el camino, la comunicación SIP punta a punta es segmentada, y el brindar nuevos servicios SIP se hace más difícil y costoso.

A modo de resumen, estas serían las recomendaciones más importantes para atravesar NAT usando SIP:

- El uso de señalización y medios simétricos en los dispositivos SIP es imprescindible.
- El intervalo de registro debe ser menor que el tiempo de expiración de los mapeos en el NAT.
- Se debe mantener vigente el mapeo en el NAT desde el dispositivo final, enviando mensajes OPTIONS o REGISTER a intervalos de tiempo cortos al registrador SIP, para permitir que el mismo mantenga las conexiones activas.
- Implementar ICE en los dispositivos SIP.
- No usar servidores STUN; los mismos no son confiables en todos los escenarios.
- Distribuir los *relays* de medios geográficamente cerca de los suscriptores.
- No usar Controladores de Sesión de Frontera para resolver sólo problemas de NAT, ya que los mismos cortan la comunicación SIP punta a punta.

2.3. H.323

La serie de recomendaciones H.323 de la ITU-T ha evolucionado a partir del trabajo de dicho organismo para estandarizar conferencias multimedia. Luego de completar la estandarización para videoconferencias en N-ISDN hasta 2Mbit/s (H.320), la ITU-T concentró sus esfuerzos en comunicaciones multimedia sobre redes ATM (H.310, H.321), sobre la PSTN utilizando módems (H.324) y sobre Ethernet isócrona (H.322). La infraestructura de red más ampliamente adoptada y por lo tanto aquella que presentaba mayores perspectivas fue atacada a partir de 1995 por H.323: las redes de área local (LAN), enfocándose en IP como protocolo de capa de red.

Por lo tanto, se podría decir que el alcance de H.323 incluye la comunicación multimedia sobre redes IP, haciendo un especial énfasis en la interconexión con las redes basadas en conmutación de circuitos.

2.3.1. Componentes del sistema

Las comunicaciones H.323 se dan entre los siguientes elementos:

Terminal: terminales son aquellos puntos finales capaces de manejar el conjunto de protocolos establecidos en las recomendaciones H.323 para comunicaciones multimedia. Pueden estar implementados puramente en software corriendo sobre una PC o ser simplemente elementos hardware especialmente creados para cumplir con dichas funciones (como teléfonos IP).

Gateway: los *gateways* proporcionan *interworking* con tecnologías que no son H.323, como videoconferencias ISDN H.320 o redes telefónicas tradicionales. Debe notarse que los *gateways* administran 1) la conversión de señalización de llamada, 2) la conversión de señalización de medios y 3) la conversión de medios cuando se conecta una red H.323 a otra de distinto tipo.

Controlador Multipunto (MC): un MC es una entidad lógica que permite interconectar la señalización de llamada y los canales de control de conferencias de dos o más entidades H.323. El MC coordina los aspectos de control para el intercambio de medios entre todas las entidades involucradas en la conferencia; provee a los puntos finales una lista de participantes y otras diversas funciones para el control de conferencias.

Procesador Multipunto (MP): para conferencias multimedia en H.323, puede utilizarse opcionalmente el MP que recibe los flujos de medios de los distintos participantes, los combina mediante alguna técnica específica de tratamiento digital y retransmite el flujo resultante a cada uno de los puntos finales.

Unidad de Control Multipunto (MCU): en el mundo H.323, un MCU es simplemente una combinación de un MC y un MP en un dispositivo único. La función principal del MCU es permitir conferencias multimedia multipunto a partir de un conjunto de conexiones punto a punto.

Gatekeeper: como su nombre lo indica, un *gatekeeper* H.323 controla y administra una zona H.323. De la misma manera que los agentes de migraciones, que controlan la entrada y salida de pasajeros a un país, un *gatekeeper* H.323 regula los puntos finales dentro de su zona que pueden iniciar o recibir llamadas. Un *gatekeeper* H.323 puede también regular el procedimiento o enrutamiento de las llamadas, permitiendo la comunicación directa entre los puntos finales, o bien actuando como intermediario para transmitir la señalización de llamada.

Los *gatekeepers* no son un requisito obligatorio en las redes H.323. Las recomendaciones especifican que, cuando los *gatekeepers* están presentes, deben llevar a cabo las siguientes funciones para los puntos finales:

- *Resolución de direcciones:* el *gatekeeper* debe ser capaz de convertir los alias H.323 o E.164 en direcciones de red e identificadores de acceso al servicio de transporte (TSAP). Por ejemplo, un *gatekeeper* puede recibir una solicitud de llamada desde un terminal para `sip&*@fing.edu.uy` o +598-2-7110974. El *gatekeeper* debe convertir estas direcciones a una dirección IP (como 164.73.38.240) y un número de puerto TCP o UDP (como el puerto TCP 1720 para el establecimiento de la conexión H.225.0).
- *Control de admisión y ancho de banda:* el *gatekeeper* es el encargado de autorizar que los terminales, *gateways* y MCU puedan realizar llamadas en la red. Dicha negociación se realiza mediante el protocolo H.225.0 RAS en una conexión que se establece entre cada entidad H.323 y el *gatekeeper*. El *gatekeeper* genera mensajes de respuesta (confirmación de admisión o rechazo) ante las solicitudes recibidas. Puede manejar diversas políticas para la toma de esta decisión o aceptar todas las solicitudes. El *gatekeeper* también se encarga de dar acceso al ancho de banda solicitado por los puntos finales y de administrar el ancho de banda y recursos disponibles en la zona H.323 que regula.

A continuación se listan algunas funciones adicionales que pueden proveer los *gatekeepers* en una red H.323.

- Pueden ofrecer un mecanismo centralizado para administrar planes de conexión y enrutamiento de llamada de una red VoIP.
- Pueden proporcionar acceso a las funciones de autenticación, autorización y contabilidad (conocidas como AAA), esenciales en los sistemas de seguridad y facturación. La interacción entre los *gatekeepers* y otras entidades que brindan funcionalidad AAA no se encuentra especificada en las recomendaciones H.323.
- Pueden proporcionar un punto centralizado para la localización de recursos basados en políticas. Por ejemplo, un servidor de políticas puede instruir a un *gatekeeper* para que registre llamadas en base al destino, disponibilidad de ancho de banda, hora del día, etc.
- Facilitan el control de llamadas a terceros, lo cual resulta esencial en sistemas tipo *call-center*.

2.3.2. Direccionamiento

H.323 emplea un esquema de nombres independiente de la tecnología subyacente de red, pero identifica los requisitos específicos de dirección para trabajar sobre IP (protocolo de capa de red más utilizado).

Direcciones de red e identificadores TSAP

El establecimiento de la comunicación con cualquier dispositivo H.323 requiere del conocimiento de su dirección de red y un identificador TSAP. En el caso de redes IP, la dirección de red es una dirección IPv4 o IPv6, y el identificador TSAP es un número de puerto TCP o UDP para establecer la conexión de transporte.

Alias H.323

Dado que las direcciones IP e identificadores TSAP son difíciles de recordar, H.323 proporciona alias para identificar los puntos finales y conferencias multiparte (dicho alias de conferencia se resuelve mediante la dirección de red e identificador TSAP del MC de la conferencia). Una opción natural hubiese sido utilizar nombres DNS en lugar de alias H.323, pero debido a que la idea original de las recomendaciones era que se pudiese operar sobre cualquier protocolo de capa de red, se crearon los alias. Los alias H.323 pueden tener varias formas:

Cadenas alfanuméricas: `SIP & *`, `sip\&*@fing.edu.uy`, `fing.edu.uy`, cadenas arbitrarias.

Direcciones E.164: +598-2-7110974, 7110974, 123.

2.3.3. Protocolos de señalización

El protocolo H.323 ofrece servicios de comunicación multiparte, multimedia y de tiempo real sobre una red QoS existente. La arquitectura de protocolos tradicional, yace sobre la pila IP tradicional (IP, TCP, UDP). En la figura 2.5 se muestra la pila de protocolos H.323.

A lo largo de esta sección se presentarán los intercambios de mensajes para los protocolos H.323 relacionados con VoIP:

- Descubrimiento de *gatekeeper* y H.225.0 RAS
- Control de llamadas H.225.0
- Control de medios H.245

Descubrimiento de gatekeeper y H.225.0 RAS

La recomendación H.225.0 de la ITU define las interacciones entre un terminal H.323 y un *gatekeeper*. Dichas interacciones incluyen un punto final que descubre un *gatekeeper* y la señalización RAS entre un punto final y el *gatekeeper* descubierto. Como lo indica claramente la

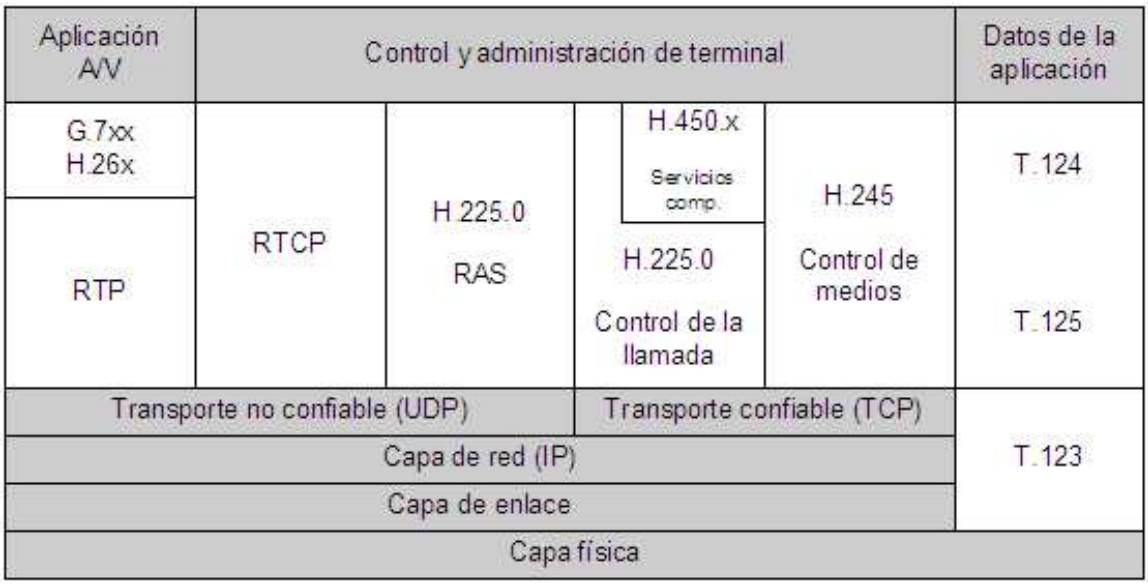


Figura 2.5: Pila de protocolos H.323

sigla RAS: *Register, Admission, State*; los terminales usan el canal RAS para registrarse en su *gatekeeper*, pedir permisos para el uso de recursos del sistema, solicitar la resolución de direcciones de terminales remotos, etc. Por otro lado, el *gatekeeper* usa RAS para mantener un registro del estado de sus terminales asociados y para recolectar información sobre los recursos disponibles una vez terminada una llamada.

La gran mayoría de los mensajes relacionados con RAS siguen una estructura común. Un punto final solicita un servicio o una acción al *gatekeeper*, que responde con una confirmación o rechazo a la solicitud. En la figura 2.6 se muestra esta transacción genérica. Cada uno de los mensajes RAS se identifica mediante una sigla de tres letras, con las siguiente formas: *RQ para las solicitudes, *CF para las confirmaciones y *RJ para mensajes de rechazo.

El caracter * en el nombre de cada mensaje genérico se reemplaza con una letra específica para representar servicios diferentes:

- Descubrimiento del *gatekeeper* (G)
- Con registro (R) y sin registro (U)
- Unificación (L)
- Admisión (A) y ancho de banda (B)
- Desacople de una llamada activa (D)

Cada uno de estos intercambios de mensajes implica el pedido de cierta información relevante para la transacción a llevarse a cabo y las respectivas respuestas dadas por el *gatekeeper*;

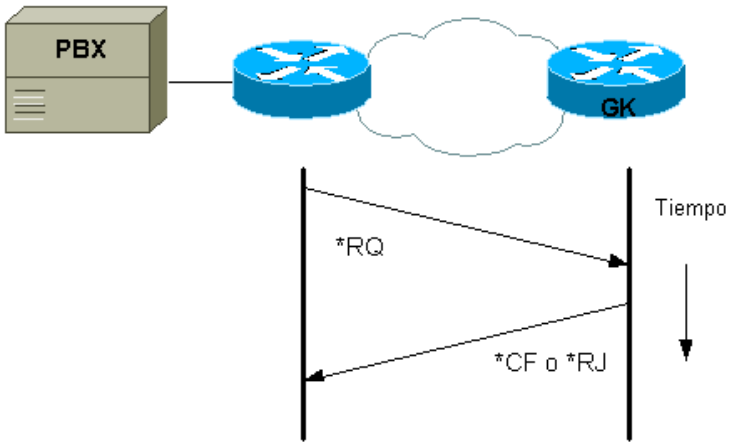


Figura 2.6: Intercambio genérico de mensajes H.225.0 RAS

y no entraremos en el detalle en cada caso. Simplemente se presentará un ejemplo integrador de intercambio de mensajes H.225.0 RAS. Para más información sobre estos temas consultar [2].

La figura 2.7 muestra un intercambio RAS típico entre puntos finales y *gatekeepers*, cuando los puntos finales se conectan a una red y cuando se desea iniciar una llamada. En la primera fase de la comunicación, el punto final intenta localizar un *gatekeeper* con un mensaje GRQ.

El *gatekeeper* responde con un GCF, indicando cómo se debe registrar. El punto final envía un mensaje RRQ al *gatekeeper*, que contiene información sobre el punto final, los alias H.323, las direcciones IP y los puertos TCP para llamadas entrantes. El *gatekeeper* acepta el registro con un mensaje RCF, que dice a los puntos finales cómo contactar al *gatekeeper* para el control de llamada H.225.0 (en caso de control de llamada enrutado por *gatekeeper*; otra posibilidad es intercambiar esta información H.225.0 directamente entre puntos finales).

Cuando un punto final desea iniciar una llamada, envía un mensaje ARQ al *gatekeeper* pidiendo permiso para hacer la llamada. Debido a que el destino se encuentra en una zona remota, el *gatekeeper* emite un LRQ al *gatekeeper* de la zona destino, que se ha determinado mediante métodos que no entran en el alcance de H.323. El *gatekeeper* remoto responde al *gatekeeper* local con un LCF que contiene las direcciones IP y el puerto TCP de destino. El *gatekeeper* local ejecuta un procedimiento de autenticación y admite al punto final con un mensaje ACF. La información de dirección que aprendió el *gatekeeper* mediante el LRQ, se envía con el ACF.

El punto final inicia el canal de control de llamada H.225.0 directamente haste el punto final de destino, lo que motiva que el destino envíe un mensaje ARQ hasta su *gatekeeper* local. Éste admite la llamada con un mensaje ACF. Al llegar a este punto, se establece el canal de control de llamada H.225.0 y, tras la negociación, los puntos finales abren un canal de control de medios H.245. En este canal, los puntos finales negocian con los puertos UDP por los flujos

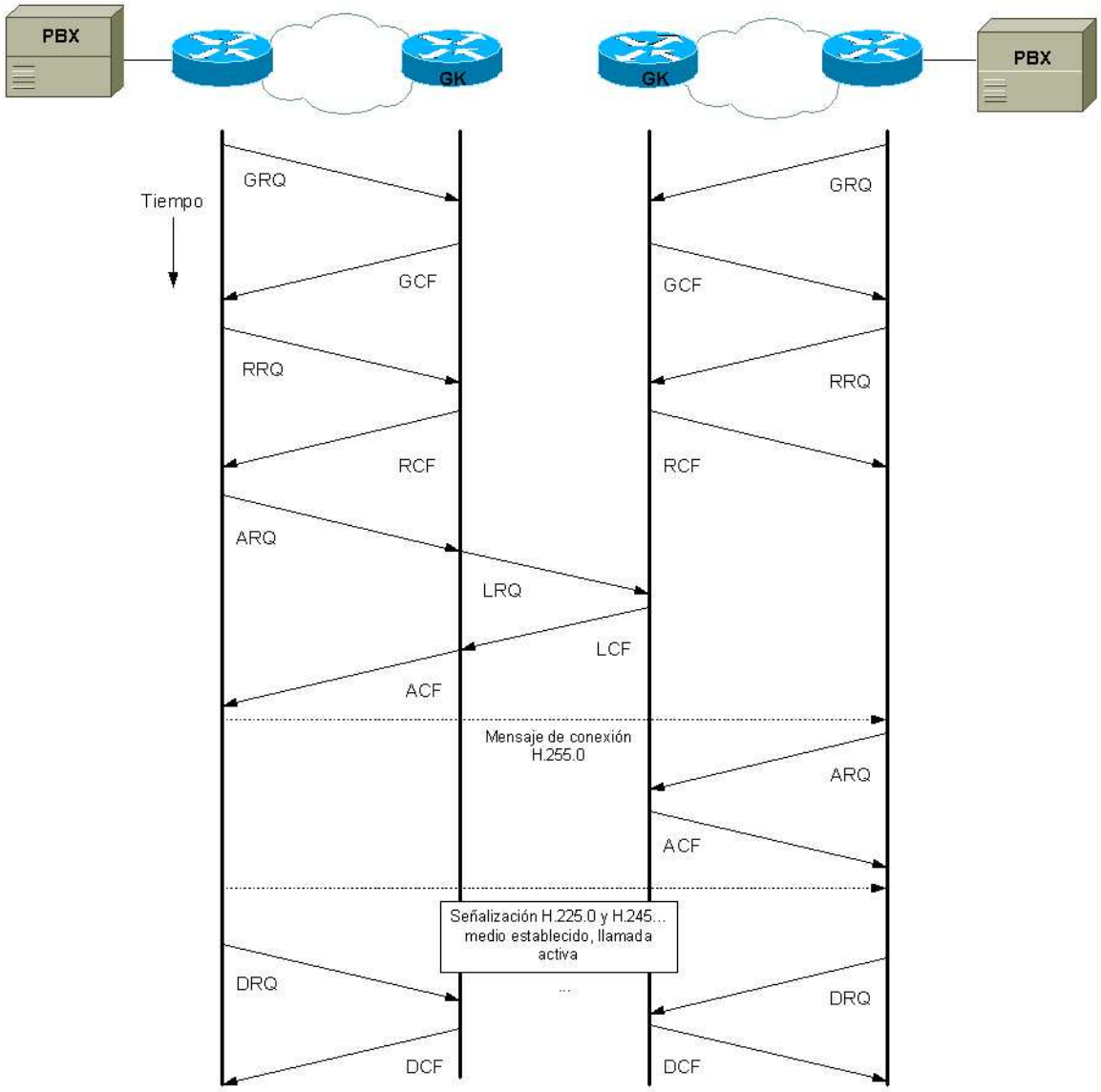


Figura 2.7: Intercambio típico de mensajes H.225.0 RAS entre puntos finales y *gatekeepers*

de audio RTP, y se establece la ruta de audio. Cuando se ha completado la llamada, cada punto final envía un DRQ a su *gatekeeper* local, mensajes reconocidos con los respectivos DCF.

Control de llamadas H.225.0

El canal de control de llamadas H.225.0 es un canal fiable por el que se intercambian conexiones de llamadas, cancelación y mensajes de servicios suplementarios. Por defecto, los puntos finales responden al puerto TCP 1720 ante solicitudes de llamadas entrantes. Los pun-

tos finales pueden recibir llamadas de un puerto TCP diferente y publicar este puerto a un *gatekeeper* para la resolución de un alias H.323. Como se observó en la figura 2.7, los puntos finales reciben la autorización (mediante un mensaje ACF) para emitir solicitudes de conexión H.225.0.

La sintaxis del mensaje del control de llamadas H.225.0 se toma directamente de las recomendaciones Q.931 y Q.932 de la ITU-T[6], aunque algunos mensajes se han eliminado y se han simplificado algunos procesos. Se buscó modernizar los intercambios engorrosos que utiliza Q.931 para la conexión y desconexión de llamadas. Q.931 asume un circuito de baja latencia, por lo que los intercambios secuenciales de mensajes no son un problema para los tiempos de conexión de llamada. H.225.0 tiene mayor presión para completar rápidamente una llamada, porque los retardos pueden ser significativos en las redes basadas en paquetes.

Enrutamiento directo vs. llamadas enrutadas por gatekeeper

El canal de control de llamadas H.225.0 en una sesión TCP puede estar enrutado directamente entre los puntos finales involucrados, o a través de uno o más *gatekeepers* con sesiones TCP separadas para cada circuito derivado de la comunicación. La decisión final corresponde al *gatekeeper* de cada punto final. Una de las principales razones por las que puede ser conveniente que el *gatekeeper* intercepte el canal de control de llamadas, es suministrar servicios *proxy*/seguridad para un punto, permitir servicios de llamada suplementarios o la capacidad de realizar conferencias entre múltiples partes. En la figura 2.8 se muestran los mensajes H.225.0 intercambiados entre las entidades en cada uno de los casos.

Control de medios H.245

El canal de control de medios H.245 se establece dinámicamente sobre una conexión TCP confiable, basándose en los parámetros del mensaje CONNECT intercambiado durante la sesión de control de llamadas H.225.0. Aunque los puertos TCP son diferentes para las conexiones H.245 y H.225.0, los puntos finales son los mismos. Los mensajes que se emplean para el control de medios H.245 se dividen en cuatro clases:

- Solicitud
- Respuesta
- Comando
- Indicación

La mayoría de las transacciones más interesantes en el control de medios H.245 se basa en secuencias solicitud-respuesta-indicación. La siguiente secuencia de acciones ocurre sobre el canal de control de medios H.245:

[6] Q.931 y Q.932 son los protocolos para el control de conexiones ISDN, pueden compararse a TCP dentro del stack TCP/IP

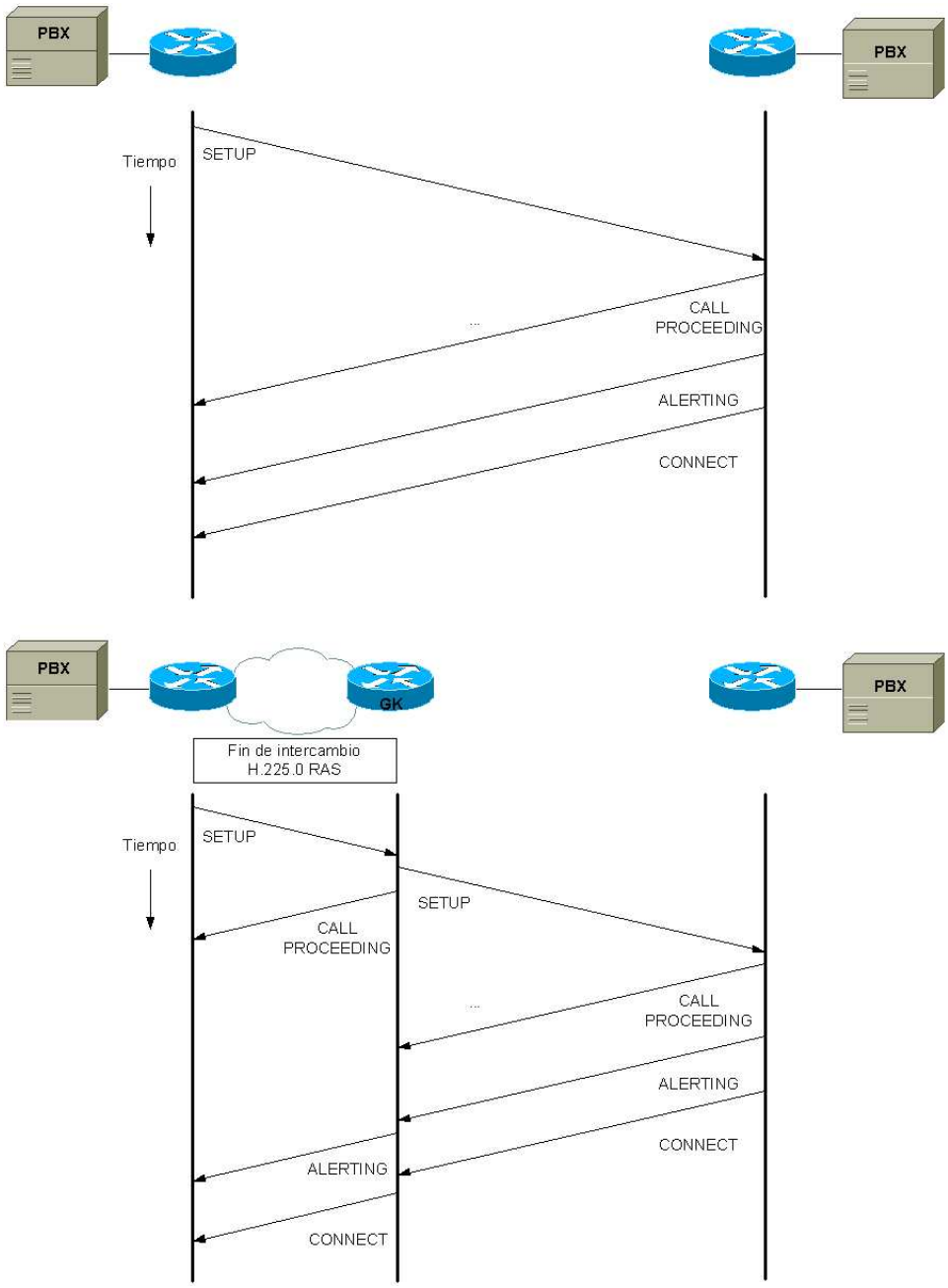


Figura 2.8: Control de llamadas H.225.0. Primero directamente entre puntos finales, luego enrutado mediante un *gatekeeper*

Intercambio de capacidades: antes de que los puntos finales puedan establecer cualquier sesión con medios, deben negociar un conjunto común de formatos de medios (codecs) que pueden usar todas las entidades H.323. La funcionalidad es similar a la del SDP en

SIP.

Determinación maestro-esclavo: uno de los puntos finales debe actuar como el maestro de la conexión H.245, encargándose de la toma de decisiones, o podrán surgir conflictos durante el establecimiento de la sesión. En general, aquel punto final que tenga mayor capacidad de medios es quien toma esta responsabilidad.

Establecimiento del canal lógico: esta fase se da cuando se inician las sesiones RTP y RTCP para el flujo de medios. Un mensaje H.245 *openLogicalChannel* inicia la sesión del flujo de medios utilizando los siguientes parámetros:

- Números de puertos UDP para sesiones RTP y RTCP
- Formato del medio (codecs de audio y video)
- Mecanismos QoS (información para las reservas RSVP)
- Otros parámetros como la transmisión o no de DTMFs, etc.

En la figura 2.9 se muestra el proceso H.245 descrito para el establecimiento del flujo de medios. Notar cómo el establecimiento de la reserva RSVP es una parte integral de la negociación entre las partes. Mediante el parámetro *flowControlToZero* se especifica a la fuente que no envíe medios hasta que culmine el proceso RSVP.

2.4. Comparación entre SIP y H.323

A lo largo de esta sección se hará una comparación entre el protocolo SIP, desarrollado por la IETF, y el protocolo H.323, desarrollado por la ITU-T.

Como se dijo en secciones anteriores, ambos protocolos se abocan a la telefonía sobre Internet, para lo cual deben comprender el establecimiento de conexión, el intercambio de capacidades y el control de conferencias.

La comparación se hará a nivel de complejidad, extensibilidad, escalabilidad y servicios.

A partir de esta comparación, quedará claro el por qué elegimos el protocolo SIP para hacer un estudio detallado en nuestro proyecto, en lugar de H.323.

2.4.1. Complejidad

En general, se puede decir que el protocolo H.323 tiene una complejidad mayor que la de SIP.

A modo de ejemplo, mientras que H.323 define cientos de elementos, SIP sólo define 37 encabezados (de los cuales 32 se encuentran en la especificación básica, y 5 en las extensiones

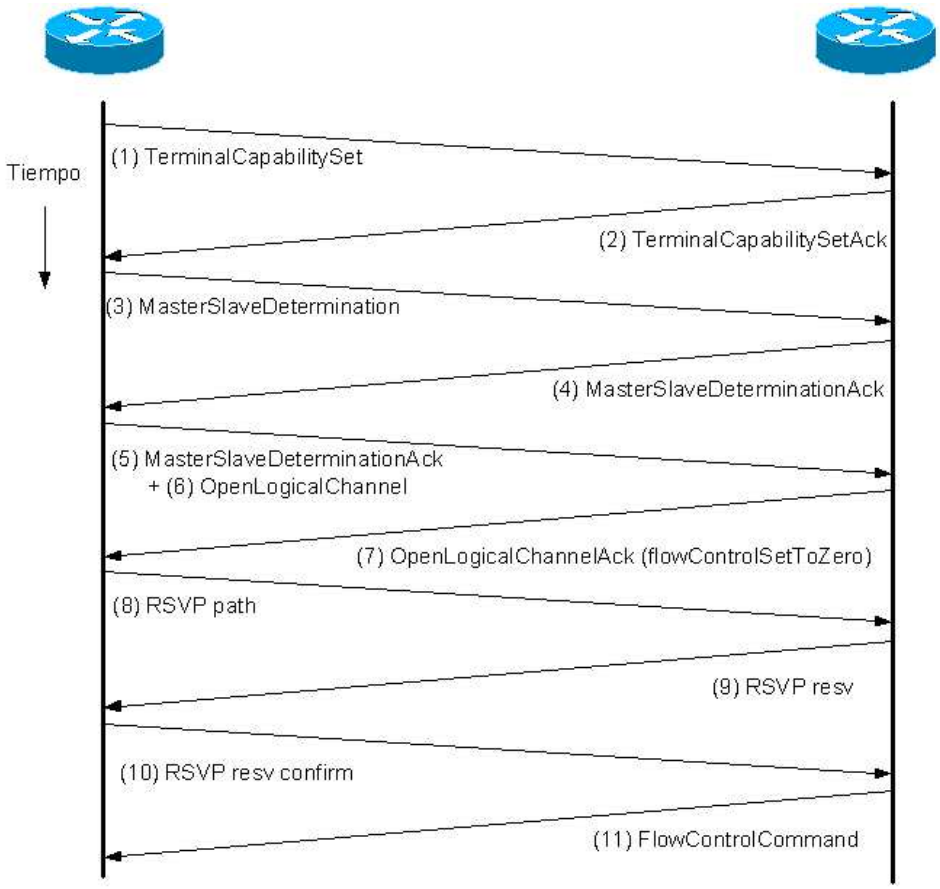


Figura 2.9: Establecimiento del flujo de medios mediante mensajes H.245.

para llamadas), cada uno con una cantidad pequeña de valores y parámetros, pero que contienen más información. Con tan solo cuatro encabezados (`To`, `From`, `Call-ID` y `C-Seq`) y tres tipos de solicitud (INVITE, ACK y BYE) puede lograrse una implementación rudimentaria pero que funciona de telefonía sobre Internet, a través de SIP.

Por otro lado, H.323 utiliza una representación binaria de sus mensajes, basada en ASN.1 y reglas de codificación de paquetes (PER). ASN.1 requiere generadores de código especiales para ser analizado. Por su parte, SIP codifica sus mensajes como texto, de forma similar a HTTP o SMTP. Esto simplifica el análisis de los mensajes y el *debugging* (incluso es posible ingresar mensajes manualmente). Su parecido con HTTP también permite la reutilización de código y analizadores existentes de HTTP, que pueden ser modificados ligeramente para ser usados con SIP.

Otro aspecto relativo a la complejidad de H.323 tiene que ver con su uso de componentes de varios protocolos. No hay una clara separación entre estos componentes, y muchos servicios requieren la interacción de varios de ellos. El uso de varios protocolos también complica el

pasaje a través de *firewalls*, aún más que en el caso de SIP.

A su vez, H.323 provee una diversidad de métodos y opciones diferentes para llevar a cabo hasta las tareas más simples. Por ejemplo, existen tres maneras distintas en que H.225.0 y H.245 pueden interactuar: el enfoque original H.323v1 de conexiones separadas, el *tunneling* de H.245 a través de H.225.0, y FastStart en H.323v2. Como H.323 permite cualquiera de los tres enfoques, los *firewalls*, sistemas finales, *gatekeepers* y *gateways* deben soportarlos a todos.

Un aspecto adicional en la complejidad de H.323 consiste en que funcionalidades presentes en otras partes del protocolo son duplicadas. En particular, H.323 hace uso de RTP y RTCP. RTCP ha sido previsto para permitir control sobre conferencias de cualquier cantidad de usuarios, es decir, de pequeña, mediana y gran escala. A su vez, H.245 prevé el control sobre conferencias de pequeña y mediana escala. De esta forma, puede verse que estos mecanismos de H.245 son redundantes.

2.4.2. Extensibilidad

La extensibilidad es una métrica clave para medir un protocolo de señalización de telefonía sobre IP. Dado el crecimiento de este servicio, es imprescindible que exista compatibilidad entre versiones ante las distintas mejoras que van surgiendo a medida que nuevas aplicaciones se van desarrollando.

Como se mencionó anteriormente en este documento, SIP ha aprendido de HTTP y SMTP (los cuales son hoy muy usados y han evolucionado a lo largo del tiempo), y por ende fue construido sobre funciones ricas en extensibilidad y compatibilidad. Por defecto, valores y encabezados desconocidos son ignorados. A través del encabezado `Require`, los usuarios pueden indicar aspectos que el servidor debe entender. Si alguno de ellos no es soportado, el servidor devuelve un código de error diciendo cuál es (o cuáles son). De esta forma el cliente puede identificar el aspecto problemático, eliminarlo y seguir trabajando normalmente. A su vez, nuevos nombres pueden ser registrados en la IANA, manteniéndose así la compatibilidad entre versiones aún cuando sean distintos desarrolladores quienes hayan registrado esos nombres.

Para mejorar aún más la extensibilidad, la identificación numérica de los mensajes está organizada jerárquicamente, como en HTTP. Hay seis clases básicas, identificadas por el dígito de las centenas del código. El resto de los dígitos brindan información adicional acerca del error.

Como los encabezados llevan nombres descriptivos (`To` - De, `From` - A, etc.), a medida que se agregan nuevos encabezados, no es difícil darse cuenta de su funcionalidad. Este tipo de estandarización no documentada ni distribuida ha sido muy común en el desarrollo de SMTP.

Dada la similitud entre HTTP y SIP, varios mecanismos que se han desarrollado para permitir la extensibilidad del primero pueden aplicarse al segundo. Entre ellos se encuentra el Protocolo de Extensión de Protocolos (PEP), que contiene punteros a la documentación de diversos

aspectos dentro del propio mensaje HTTP.

Por su parte, H.323 también provee mecanismos de extensibilidad. Son generalmente campos `nonStandardParam`, situados en diversas posiciones del ASN.1. Este parámetro contiene un código de fabricante, seguido de un valor opaco que solamente tiene sentido para ese fabricante. Esto le permite a diferentes fabricantes desarrollar sus propias extensiones. Sin embargo, tiene sus limitaciones. Primero que nada, las extensiones sólo se limitan a campos donde haya sido agregado algún `nonStandardParam`. En segundo lugar, H.323 no tiene mecanismos para permitirle a las terminales intercambiar información sobre cuáles extensiones soportan, lo cual limita la interoperabilidad entre terminales de diferentes fabricantes.

A su vez, H.323 requiere completa compatibilidad hacia atrás, entre una versión nueva y las otras más antiguas. Como surgen funcionalidades nuevas, y otras desaparecen, lo único que ocurre con el código es que crece. Sin embargo, SIP permite que encabezados y funcionalidades viejas desaparezcan gradualmente hasta no ser más requeridos, manteniendo el protocolo y su codificación concisa y limpia.

Un asunto crítico para la extensibilidad son los codecs de audio y video. Cientos de codecs han sido desarrollados, de los cuales varios son propietarios. SIP utiliza SDP para conocer qué codecs soportan los usuarios finales en una sesión. Los codecs son identificados por nombres que pueden ser registrados en la IANA, y luego usados. Esto implica que SIP puede trabajar con cualquier codec y otras implementaciones pueden determinar el nombre del codec y su información gracias a la IANA.

En H.323 cada codec debe ser registrado y estandarizado. Actualmente sólo los codecs de la ITU pueden ser usados (G.711A, G.771μ, G.729, G.726, entre otros). Como muchos de ellos no son gratuitos, se presenta una barrera para el uso de H.323 de parte de universidades y usuarios normales.

A su vez SIP permite que nuevos servicios sean definidos a partir de unos pocos mecanismos de control de terceros, muy poderosos. Estos mecanismos le permiten a un tercero instar a otra entidad a que cree y destruya llamadas de otras entidades. A medida que la instancia controlada ejecuta las instrucciones, al controlador le son enviados mensajes de estado, lo cual le permite tomar acciones basándose en la ejecución local de un programa. Como hay cientos de servicios de telefonía actualmente ya definidos, no es razonable intentar escribir especificaciones para cada uno. SIP permite que estos servicios sean utilizados basándose en mecanismos simples y estándar. Estos mecanismos pueden ser usados para construir una gran variedad de servicios, como transferencia asistida, transferencia incondicional, llamadas de tres usuarios, etc. En la Sección 5.4 se hace un estudio detallado de la implementación del servicio de transferencias, analizando las recomendaciones de la IETF para su implementación en SIP.

Otro aspecto de la extensibilidad es la modularidad. La telefonía sobre IP requiere un número muy grande de funciones distintas; entre ellas se encuentran la señalización básica, el control

de conferencias, la calidad de servicio, el acceso a directorios, el descubrimiento de servicios, etc. Es seguro que a lo largo del tiempo los mecanismos que llevan a cabo estas funciones cambiarán (especialmente aquellos que tienen que ver con la calidad de servicio). Por lo tanto, es crítico que dichas funciones puedan ser estructuradas en componentes separados, modulares, ortogonales, que puedan ser intercambiados a lo largo del tiempo. También es crítico utilizar protocolos separados y generales para cada una de estas funciones.

SIP es razonablemente modular. Abarca señalización básica, localización de usuarios y registro. La señalización avanzada es parte de SIP, pero dentro de una única extensión. La calidad de servicio, el acceso a directorios, el descubrimiento de servicios, la descripción de contenido de sesión y el control de conferencias son ortogonales, y residen cada uno en un protocolo diferente. Por ejemplo, es posible utilizar la capacidad de descripción de H.245 dentro de SIP, sin hacerle ningún cambio al protocolo SIP en sí.

Por el contrario, H.323 es menos modular. Define un conjunto de protocolos integrados verticalmente para una única aplicación. La mezcla de servicios provista por los componentes de H.323 abarca el intercambio de capacidades, control de conferencias, operaciones de mantenimiento, señalización básica, calidad de servicio, registro y descubrimiento de servicios. A su vez, todos estos servicios están entrelazados a través de los múltiples sub-protocolos dentro de H.323.

La modularidad de SIP le permite ser usado en conjunto con H.323. Un usuario puede usar SIP para ubicar a otro usuario, sirviéndose de su facilidad para búsquedas multi-salto. Cuando el usuario finalmente ha sido localizado, pueden usar una respuesta de redirección a una URL H.323, indicando que la comunicación real deberá llevarse a cabo bajo el protocolo H.323.

2.4.3. Escalabilidad

También se observan diferencias entre SIP y H.323 en términos de escalabilidad. La escalabilidad se puede ver a diferentes niveles:

Alto número de dominios: H.323 fue originalmente concebido para ser usado en LANs, donde asuntos como el direccionamiento global y ubicación de usuarios no eran un problema. Las nuevas versiones definen el concepto de "zona", y procedimientos para la ubicación de usuarios a través de las mismas. Sin embargo, para un alto número de dominios, y operaciones complejas de ubicación, H.323 tiene problemas de escalabilidad: no provee una forma fácil de detectar bucles en búsquedas multi-dominio complejas. Por el contrario, SIP usa un algoritmo detector de bucles similar al de BGP, que puede ser implementado sin restricciones de tamaño.

Procesamiento de servidor: En un sistema H.323, los *gatekeepers* de telefonía serán necesarios para manejar las llamadas de múltiples usuarios. En el esquema SIP, serán necesarios los servidores *proxies* SIP. Para proveedores IP grandes, de backbone, el número de llamadas a manejar puede ser significativo.

En SIP, una transacción a través de varios servidores *proxies* puede hacerse de manera *stateless* o *stateful*. En el modelo *stateless*, un servidor recibe una solicitud de llamada, lleva a cabo alguna operación, reenvía la solicitud y se olvida completamente de ella. Los mensajes de SIP contienen la suficiente información de estado como para permitir que la solicitud sea reenviada correctamente. A su vez, SIP puede ser transportado tanto sobre TCP como sobre UDP. En el caso de UDP, no se requiere establecimiento de conexión. Esto implica que los proveedores grandes, de backbone, pueden utilizar un modelo *stateless* sobre UDP reduciendo ampliamente los requerimientos de memoria y mejorando la escalabilidad.

Por otra parte, H.323 requiere *gatekeepers* que deben ser *stateful*; estos deben mantener el estado de la llamada por toda su duración. Además, las conexiones se basan en TCP, lo cual implica que los *gatekeepers* también deben mantener las conexiones TCP durante toda la llamada. Esto puede traer serios problemas de escalabilidad para *gatekeepers* grandes.

A su vez, los *gatekeepers/proxies* deberán procesar los mensajes de señalización para cada llamada. Cuanto más simple sea la señalización, más rápido podrá ser procesada, y más llamadas podrá manejar el *gatekeeper/proxy*. Como SIP es más simple de procesar que H.323, debería permitir procesar más llamadas por segundo que H.323 (en un mismo dispositivo).

Tamaño de conferencias: H.323 soporta conferencias multiusuario con distribución de datos multicast. Sin embargo, requiere un punto de control central (llamado MC) para procesar toda la señalización, hasta para las conferencias más reducidas. Esto tiene varios inconvenientes. Primero que nada, si el usuario que brinda la funcionalidad de MC deja la conferencia, y sale de la aplicación, entonces la conferencia se cae. Además, como las funcionalidades de MC y *gatekeeper* son opcionales, en algunos casos H.323 puede no soportar siquiera conferencias de tres usuarios. Se puede notar que el MC es un cuello de botella para conferencias grandes. Para aliviar esto, la nueva versión de H.323 prevé el uso de MCs en cascada, permitiendo un árbol de distribución multicast de control de mensajes un tanto limitado. Esto mejora en algo la escalabilidad, pero aún para conferencias mayores, el protocolo H.332 define procedimientos adicionales. Esto implica que existen tres mecanismos diferentes para soportar conferencias de distintos tamaños.

SIP, por otro lado, escala a conferencias de todos los tamaños posibles. No se requiere un MC central; la coordinación de la conferencia está distribuida. Esto representa mejoras tanto en escalabilidad como en complejidad. Además, como SIP puede usar tanto UDP como TCP, soporta la señalización multicast nativa, permitiéndole a un protocolo escalar entre sesiones de dos a millones de miembros.

Retorno (Feedback): H.245 define procedimientos que permiten a los receptores controlar la codificación de medios, tasas de transmisión y recuperación de errores. Este tipo de

retorno tiene sentido en escenarios punto a punto, pero deja de ser funcional en conferencias multipunto. SIP, por su lado, se apoya en RTCP para brindar retorno en cuanto a la calidad de recepción (y también para obtener listas de membrecías de grupos). RTCP, como SIP, opera en una forma completamente distribuida. El retorno que provee escala automáticamente de una conferencia punto a punto de dos usuarios a una conferencia del tipo broadcast con millones de usuarios.

2.4.4. Servicios

SIP y H.323 ofrecen aproximadamente los mismos servicios. Sin embargo, es difícil hacer una comparación exacta ya que continuamente surgen nuevos servicios para cualquiera de los dos protocolos.

Además de los servicios de control de llamadas, tanto SIP (cuando es usado con SDP) como H.323 proveen capacidades de intercambio de servicios. En este sentido, H.323 tiene una funcionalidad mucho más rica. Los terminales pueden expresar su habilidad de utilizar diferentes codificaciones y decodificaciones basándose en parámetros de los codecs y en cuáles otros codecs están en uso. Sin embargo, la mayoría de las implementaciones no implementan esto, y la capacidad básica de recepción que brinda SIP ("elija cualquier conjunto de los siguientes codecs para esta lista de flujos de medios") parece suficiente y equivalente a las funcionalidades de H.323 realmente implementadas.

Por otro lado, el soporte de SIP a servicios de movilidad es muy rico. Cuando el usuario llamante contacta al llamado, el llamado puede redirigirlo a un sinnúmero de ubicaciones diferentes. Cada una de estas nuevas ubicaciones puede ser una URL arbitraria, y contiene información adicional sobre el terminal en esa ubicación, como el idioma que se habla, si se trata de una residencia o una oficina, un teléfono fijo o móvil, y una lista de prioridades del llamado. Esto le da la flexibilidad suficiente al llamante para decidir con qué ubicación quiere hablar. Para terminales no interactivas, el establecimiento original de la llamada puede tener en cuenta las preferencias del usuario llamante sobre a qué tipo de ubicación quiere llamar. De esta forma, los *proxies* pueden basarse en este tipo de requisitos para reenviar llamadas o no hacerlo.

SIP también soporta búsquedas multisalto de usuarios. Cuando se hace una solicitud de llamada a determinada ubicación, un servidor SIP es contactado en esa dirección. Como este servidor SIP puede no estar ubicado en la misma máquina en que está realmente el usuario llamado, puede reenviar la llamada a uno o más servidores adicionales. Estos servidores pueden a su vez seguir reenviando la solicitud hasta encontrar al usuario llamado. Un servidor puede también reenviar la solicitud a varios servidores en paralelo, lo cual hace que la búsqueda sea más rápida. SIP también permite que múltiples ramas de la búsqueda acepten la llamada, enviándole las respuestas de vuelta al usuario llamante, quien decidirá con quién hablar.

El soporte de H.323 para una movilidad de este tipo es bastante más limitado. El mensaje "Fa-

cility" puede redirigir al usuario llamante a intentar con otras direcciones. Sin embargo, no puede ser utilizado para expresar preferencias, ni de parte del usuario llamado ni del usuario llamante. H.323 no fue desarrollado para operaciones de área amplia; permite sí el reenvío de solicitudes de llamadas entre servidores, pero no tiene mecanismos de detección de bucles. H.323 tampoco permite que un *gatekeeper* reenvíe una solicitud a varias direcciones destino en paralelo.

H.323 soporta varios servicios de control de conferencia y resolución de participantes. SIP no provee control de conferencia, recayendo en otros protocolos para estas funciones. Sin embargo algunos mecanismos simples de control de conferencia (como el envío de notas entre participantes, y el poder obtener una lista de los participantes activos), están disponibles a través de RTCP.

2.4.5. Conclusiones

Como se pudo ver a través de todos estos puntos, SIP provee un conjunto de servicios similar al de H.323, pero con una complejidad sustancialmente menor, una rica extensibilidad y mejor escalabilidad.

Es por todas estas razones por las que decidimos ahondar en el estudio de SIP a lo largo de nuestro proyecto, en lugar de H.323.

2.5. Introducción a Asterisk

2.5.1. ¿Qué es Asterisk?

Muchos dicen que Asterisk (de aquí en más simplemente *) es la navaja suiza de la telefonía. Oficialmente * es una PBX (Private Branch Exchange) híbrida totalmente implementada en software: maneja tanto TDM como voz paquetizada. Uno de sus atributos más importantes es que se trata de una solución Open Source bajo el GNU General Public License, por lo cual el administrador puede diseñar su sistema y modificar las funcionalidades implementadas de acuerdo a las necesidades de su aplicación. Esto permite que día a día el sistema sea mejorado por la comunidad de usuarios, lo que garantiza que * está preparado para madurar ante la demanda del mercado de telecomunicaciones.

* corre bajo Linux, FreeBSD y MacOS X[7], y provee todas las funcionalidades de una PBX. Permite hacer VoIP en al menos 5 protocolos (MGCP, SIP, H.323, IAX y Skinny), y puede interoperar con casi todos los equipos de telefonía basados en estándares, usando hardware de adaptación por un costo relativamente bajo. Tradicionalmente los productos para telefonía se caracterizaban por satisfacer una necesidad técnica particular dentro de una red. Sin embargo, cada vez más las aplicaciones de voz en tiempo real involucran diversas tecnologías en conjunto, por lo cual las soluciones tradicionales resultan ser prohibitivamente costosas

[7] Actualmente existen proyectos para correr * bajo Windows como *AstWind* [33] y AsteriskWin32 [34], pero en ambos casos con funcionalidad muy limitada.

en gran cantidad de situaciones. Dentro de esta realidad, * permite crear un único escenario que puede ser moldeado para satisfacer cualquier aplicación, o conjunto de aplicaciones, de acuerdo a las necesidades del usuario.

Podemos resumir los objetivos principales de * en los siguientes puntos:

- Proveer implementaciones Open Source de la funcionalidad básica de PBX telefónicas escalando desde soluciones SOHO (Small Office Home Office) a grandes escenarios empresariales con la posibilidad de interconectar diversos sistemas * en puntos remotos.
- Ser flexible para el diseño de funcionalidades avanzadas.
- Permitir la migración del hardware propietario a software Open Source.
- Ser independiente de los distintos fabricantes de hardware creando un escenario de interoperabilidad de las distintas soluciones.
- Proveer una colección de módulos para la implementación de servicios de telefonía.

Respondiendo a la pregunta planteada, * esencialmente es:

- PBX telefónica - canales TDM: PSTN, PRI, BRI, etc.
- Gateway VoIP - canales IP.
- Framework para el desarrollo de aplicaciones y servicios de telefonía.
- IVR (Interactive Voice Response).
- Plataforma de correo de voz.
- Diversas funciones que harían este listado interminable.

2.5.2. Tecnologías soportadas

* busca integrar fácilmente nuevas interfaces y tecnologías. En general, las interfaces manejadas pueden dividirse en tres grandes categorías: interfaces Zaptel, interfaces no-Zaptel y voz sobre tecnología de paquetes.

Interfaces Zaptel

Estas interfaces permiten la integración con los sistemas tradicionales de telefonía analógica y digital. Soportan entre ellas lo que se conoce como conmutación pseudo-TDM permitiendo una latencia casi nula entre llamadas y conferencias TDM. Este hardware es la base del negocio de los creadores de *, que a través de la empresa Digium vende las tarjetas Zaptel. Dichas tarjetas permiten conectar * con diversas interfaces de red como PSTN, T1, E1, PRI, E&M, entre otras. Originalmente, por razones técnicas era necesario tener al menos una interfaz Zaptel configurada para correr ciertas aplicaciones, como por ejemplo las conferencias multiparte. Hoy esto ya no es cierto, ya que por software se ha podido implementar aquella funcionalidad que antes solamente brindaban las tarjetas.

Interfaces no-Zaptel

Son interfaces que también permiten la integración con los sistemas telefónicos tradicionales. Sin embargo, no soportan la conmutación pseudo-TDM. Como un ejemplo se tienen las interfaces ISDN BRI compatibles con los stacks ISDN para Linux; CAPI y mISDN.

Protocolos de voz sobre paquetes

* maneja la mayoría de los protocolos de VoIP en uso actualmente. Son las únicas interfaces que no precisan de hardware alguno para operar. Entre los protocolos soportados se encuentran: SIP, MGCP, H.323 e IAX. IAX (*Inter Asterisk eXchange*) es el protocolo de VoIP propietario de *, el cual se ha diseñado específicamente para el transporte de voz. Fue pensado para trabajar en escenarios con NAT y ser muy eficiente desde el punto de vista del consumo de ancho de banda.

2.5.3. Arquitectura de Asterisk

La arquitectura de * es en esencia muy sencilla, pero se diferencia bastante de aquella utilizada por los productos tradicionales para telefonía. * se encuentra enmarcado en una posición central, conectando diferentes tecnologías telefónicas en la parte baja de la arquitectura con aplicaciones en la parte superior. Esta arquitectura es ideal para el desarrollo de escenarios mixtos, donde diversas tecnologías conviven e interoperan de acuerdo a las necesidades del usuario. Como ejemplo de posibles tecnologías manejadas podemos citar: SIP, H.323, IAX e interfaces tradicionales como E1, T1, conexión a la PSTN, ISDN PRI, entre otras. Dentro de las aplicaciones usuales tenemos: conmutación de llamadas entre diferentes interfaces, conferencias, correo de voz, scripting IVR, transferencia asistida de llamadas, grabación de audio, consultas a bases de datos, *text2speech* y muchas otras más.

Considerando que un importante objetivo del sistema es la flexibilidad, se ha buscado que los elementos centrales de la arquitectura encargados de manejar las interconexiones de la PBX puedan abstraerse completamente de los diferentes protocolos, codecs, o drivers para las interfaces hardware utilizados en las diversas aplicaciones de telefonía. Por otro lado, se han definido un conjunto de APIs específicas para establecer la interacción entre el núcleo y las diferentes aplicaciones o módulos que implementan las interfaces de comunicación.

El núcleo de * maneja los siguientes elementos internamente:

1. **Conmutación de llamadas** - la esencia de * es implementar una PBX para la conmutación de llamadas telefónicas. El núcleo de conmutación permite interconectar de manera transparente diferentes tecnologías, por ejemplo operando como *gateway* entre terminales SIP y la PSTN.

2. **Ejecución de aplicaciones** - permite ejecutar las diferentes aplicaciones cargadas como módulos dinámicos y que proveen diversos servicios a los usuarios como el correo de voz.

3. **Traducción de codecs** - manejando diversos módulos independientes permite la codificación/decodificación de los formatos de audio más comúnmente utilizados en la industria telefónica.

4. **Despachador y administrador de E/S** - maneja las tareas de scheduling y administración de más bajo nivel en el sistema, garantizando óptima performance con múltiples niveles de carga.

Asimismo se definen cuatro grandes APIs para los módulos dinámicos que pueden cargarse al sistema, de manera de facilitar la abstracción de hardware y protocolos. Es decir, se establecen interfaces bien conocidas entre el elemento central y los diferentes módulos pero las funcionalidades quedan completamente encapsuladas facilitando la administración y extensión del sistema.

1. **API de canales** - esta interfaz para los canales se encarga del manejo del tipo de conexión en la que llega o se genera una llamada, sea esta PSTN, mediante un protocolo de voz paquetizada como SIP o cualquier otra tecnología aplicable. Ciertos módulos dinámicos conocidos como drivers son los que se encargan del manejo de bajo nivel de estas conexiones.

2. **API de aplicaciones** - esta API es la que permite que ciertos módulos creados para realizar tareas específicas se ejecuten para cumplir con la funcionalidad requerida por la aplicación. Un ejemplo de aplicación típica son las conferencias multiusuario.

3. **API de codecs** - define la estructura para los módulos encargados de la codificación y decodificación de audio en diversos formatos como GSM, G711A, G711μ, G729, G726, iLBC, Speex, etc.

4. **API de formato de archivos** - maneja la lectura y escritura de archivos en diversos formatos para el almacenamiento de datos en el sistema.

Esto se ilustra en la figura 2.10.

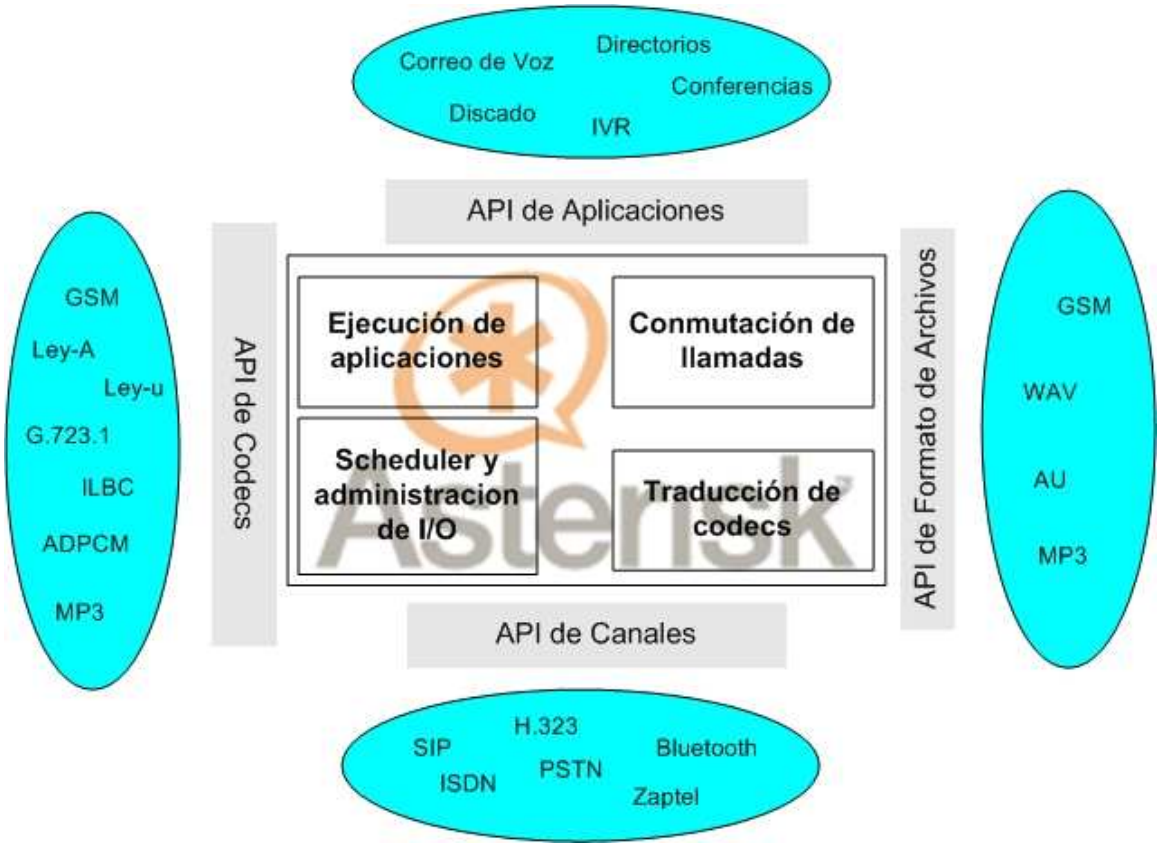


Figura 2.10: Diagrama de arquitectura de Asterisk.

3. ELEMENTOS FUNDAMENTALES DE *

Cuatro elementos fundamentales dentro de la arquitectura de * son: los **canales**, el ***dialplan*** (plan de marcación), la **interfaz de línea de comandos (CLI)** y las **aplicaciones** . A continuación se hará una descripción de estos cuatro aspectos.

3.1. Canales

Los canales son conexiones lógicas a los variados caminos de señalización, transmisión y recepción de voz que * es capaz de usar para crear y conectar llamadas. Según su naturaleza, pueden estar vinculados a un medio físico (como un puerto FXO/FXS analógico), o basados en software (como un canal IAX o SIP). Para enrutar una llamada que llega por uno de estos canales y saber cómo conectar los canales entre sí, * utiliza el *dialplan* (que se describe en la sección siguiente). Naturalmente, para que se pueda utilizar el *dialplan*, se deberán determinar los tipos de canales necesarios de acuerdo con los requerimientos de la aplicación. Por último, habrá que configurarlos específicamente para que puedan ser utilizados en el sistema.

Los canales pueden tener formatos variados: pueden corresponderse con dispositivos físicos de telecomunicaciones (como los FXO, FXS, BRI, PRI); implementar protocolos VoIP para la transmisión de voz paquetizada (SIP, IAX); o ser canales lógicos usados para ciertas funciones específicas de * (como Agent, Console, Local). A estos últimos se los conoce como *pseudo-canales*. * trata a todos estos tipos de canales como puntos de conexión que se conectan efectivamente según el *dialplan*. Es importante destacar que aunque los canales pueden variar en términos de tecnología y conectividad, * permite tratarlos a todos de forma similar lo cual constituye una de las principales ventajas del enfoque de * para el armado de una PBX. Mediante la especificación en la API de canales de una estructura común para la interfaz con el núcleo de *, se logra una completa abstracción en las funciones del sistema que operan sobre los canales, independientemente de la implementación de cada canal particular.

* es una PBX extremadamente poderosa y flexible en gran parte gracias a cómo maneja los canales. Siempre que sea posible * le permite al usuario trabajar con un tipo dado de canal de la misma forma que trabaja con cualquier otro tipo. Ya sea que se está conectado a un teléfono analógico a través de un puerto FXS, a un softphone SIP, o a un teléfono IAXtel mediante un canal IAX, las funciones disponibles a través de * serán esencialmente las mismas.

Los distintos tipos de canales se configuran cada uno en su correspondiente archivo de configuración. Los archivos de configuración de canales de * se encuentran en `/etc/asterisk`.

En ese directorio se encuentran, entre otros, los archivos `iax.conf` (para configuración de canales IAX), `sip.conf` (para configuración de canales SIP), `zapata.conf` (para configuración de canales Zaptel), etc.

En el archivo `extensions.conf` se define el *dialplan* y por lo tanto toda la lógica para el manejo de las llamadas que arriban a la PBX desde los distintos canales que fueron configurados.

En las siguientes subsecciones se mostrarán las características fundamentales de los principales canales de *: Zaptel a través de una descripción de las opciones FXS y FXO, IAX, y también se hará mención a los canales H.323 y otros canales que son manejados por *. No se incluye aquí información sobre el canal SIP de *, para el cual se realiza un estudio detallado en el Capítulo 4.

3.1.1. Zaptel - Módulos FXO y FXS

Los términos "FXO" y "FXS" tienen su origen en un antiguo servicio telefónico llamado "Foreign eXchange" (FX). El propósito original de un circuito FX era el de permitir que un teléfono analógico remoto pudiera conectarse a una PBX situada en algún otro lugar. Un circuito FX tiene dos terminales: la terminal de Estación (Station), donde está conectado el teléfono; y la terminal de Oficina (Office), donde está situada la PBX. Luego, las tarjetas FXO y FXS reciben su nombre de acuerdo a lo que es conectado a ellas: la terminal de Oficina (PBX) se conecta a una tarjeta FXO, y por lo tanto el comportamiento de la tarjeta deberá emular el de un teléfono; por otro lado, la terminal de Estación (teléfono) se conecta a una tarjeta FXS, y por lo tanto el comportamiento de la tarjeta deberá emular el de una central telefónica.

FXS

Los módulos Foreign eXchange Station (FXS) proveen la misma interfaz que provee una compañía telefónica a los hogares y oficinas a los que llega. Luego, entre otras cosas, los módulos FXS normalmente brindarán:

- Tono de discado
- Tensión de campanilla
- Detección de DTMF
- Mensaje en espera
- Identificación de línea llamante

En *, cuando se configura una tarjeta Digium FXS (como la TDM400P con tarjetas FXS instaladas), es necesario definir ciertos parámetros que son relevantes para la misma. Lo que es más importante, debe recordarse que la tarjeta FXS provee la señalización de una central

telefónica. Esta configuración se hace a través del archivo de configuración `/etc/asterisk/zapata.conf`. Un ejemplo del contenido de este archivo se muestra a continuación:

```
language=es
context=default
signalling=fxo_ks
channel => 1
```

La línea `signalling=fxo_ks` describe la funcionalidad que la tarjeta está brindando. Dado que la tarjeta FXS actúa como una central telefónica, se especifica que la señalización que se quiere brindar es la de una central telefónica (Office).

Cabe mencionar que en el ejemplo citado no se incluyen todas las posibles opciones que pueden ser configuradas para la tarjeta.

FXO

Una tarjeta FXO es aquella que se conecta a la central telefónica (o terminal de Oficina - Office -). Un módem es un ejemplo típico de una tarjeta FXO. Un dispositivo FXO debe ser capaz de:

- Generar DTMF
- Detectar tono de discado
- Detectar timbrado
- Detectar mensaje en espera
- Interpretar la identidad del usuario llamante
- Brindar la condición de colgado/descolgado al extremo remoto, así como la funcionalidad de Flash

La principal diferencia entre la configuración de una tarjeta FXO y FXS consiste en el parámetro de señalización. En el archivo `/etc/asterisk/zapata.conf`, se agrega:

```
signalling=fxs_ks
channel => 4
```

Luego, el archivo completo quedaría:

```
language=es
context=default
signalling=fxo_ks
channel => 1
signalling=fxs_ks
channel => 4
```

Los canales 1 y 4 de la tarjeta comparten los mismos atributos, salvo la señalización: en el primer caso, la misma es `fxo_ks`, y en el segundo caso, es `fxs_ks`.

Finalmente, al canal 1 se le podrá conectar un teléfono analógico, mientras que al canal 4 se le podrá conectar el circuito analógico provisto por la compañía telefónica.

3.1.2. IAX

El protocolo *Inter Asterisk eXchange* (IAX o IAX2) es un protocolo de transporte de medios basado en IP. Mediante este protocolo, se puede:

- Conectar servidores * entre sí
- Conectarse a la red IAXtel[1]
- Conectar clientes IAX

Para crear un canal IAX, deben definirse los parámetros correctos dentro del archivo de configuración `/etc/asterisk/iax.conf`.

Parámetros generales

Antes que nada, deben establecerse los parámetros generales que usarán los canales IAX. A continuación se muestran algunos de los parámetros que pueden definirse.

```
[general]

port=4569           ;Puerto al que conectarse (4569 es el puerto
                    ;por defecto para IAX2)
bindaddr=0.0.0.0    ;Interfaces de red del
                    ;servidor a las cuales conectarse
deny=all            ;Deshabilita todos los codecs, para poder
                    ;habilitar específicamente aquellos que serán usados
allow=ulaw
```

[1] Red de prueba que le permite a clientes IAX y usuarios de * conectarse entre sí; es un servicio ofrecido por Digium.

```
allow=alaw
allow=gsm
```

Este ejemplo es muy básico y define los mínimos parámetros necesarios para poder crear y monitorear conexiones con canales IAX. Dada esta configuración, * escuchará en el puerto 4569 los posibles pedidos de conexión de un canal IAX.

Cabe notar que en el ejemplo previo, se observa que como en todo archivo de configuración de *, en `iax.conf` se pueden incluir comentarios luego del caracter `;`.

Definición de canales IAX

Una vez definidos los parámetros generales que tendrá nuestra interfaz IAX hacia el resto del mundo, estamos en condiciones de crear canales IAX. Los canales IAX son muy flexibles, y pueden ser conectados a distintos tipos de terminales.

A pesar de no estar definido en ninguna RFC, este protocolo está bastante difundido en aplicaciones de VoIP, principalmente en aquellas que utilizan * como uno de sus componentes. A continuación se muestra un ejemplo de cómo se definen los canales IAX para poder conectarnos a la red IAXtel.

```
[general]

port=4569
bindaddr=0.0.0.0
allow=all
register => username:secret@iaxtel.com  ;Se sustituye username:secret por la
                                        ;identificación brindada por IAXtel

[iaxtel-in]

type=user       ;Sección de configuración para conexiones entrantes
context=default ;Las conexiones entrantes irán a este contexto en
                ;extensions.conf
username=username
secret=password
deny=0.0.0.0/0.0.0.0    ;Por defecto se deniega el servicio
permit=216.207.245.47/255.255.255.255   ;Servidor IAXtel
permit=69.73.19.178/255.255.255.255     ;Servidor IAXtel

[iaxtel-out]

type=peer   ;Sección de configuración para conexiones salientes
username=username
secret=password
deny=0.0.0.0/0.0.0.0    ;Por defecto se deniega el servicio
permit=216.207.245.47/255.255.255.255   ;Servidor IAXtel
```

```
permit=69.73.19.178/255.255.255.255     ;Servidor IAXtel
```

Mediante la línea `register => username:secret@iaxtel.com`, indicamos que * se registre al servidor para permitir que la red IAXtel, y por ende, todos sus usuarios, conozcan nuestra ubicación actual.

Puede verse que existen dos tipos distintos de conexiones a IAXtel: `peer` y `user`. Esto nos permite diferenciar el servidor del cual recibiremos llamadas, del servidor al que le enviaremos nuestras llamadas (aunque en este ejemplo ambos servidores son el mismo). Esto es de utilidad cuando se está manejando una red grande de servidores * y se utiliza el protocolo IAX para interconectar los servidores.

Nombre de canales IAX

El formato del nombre de un canal IAX para una conexión **saliente** es el siguiente:

```
IAX/[<user>[:<secret>]@]<peer>[:<portno>][/<exten>[@<context>][/<options>]]
```

donde:

- `<user>`: UserID del *peer* remoto, o nombre del cliente configurado en `iax.conf` (opcional)
- `<secret>`: Palabra clave (opcional)
- `<peer>`: Nombre del servidor al cual conectarse
- `<portno>`: Puerto para conectarse en el servidor (opcional)
- `<exten>`: Extensión en el servidor * remoto (opcional)
- `<context>`: Contexto a usar en el servidor * remoto (opcional)
- `<options>`: La única opción permitida es `a`, que significa *solicitar auto-respuesta* (opcional)

El formato del nombre de un canal IAX para una conexión **entrante** es el siguiente:

```
IAX/[[<username>@]<host>]/<callno>
```

donde:

- `<username>`: el nombre de usuario, si es conocido (opcional)
- `<host>`: El servidor que se está conectando (opcional)
- `<callno>`: El número de llamada local

3.1.3. H.323

El canal H.323 de * se incluye en el código de distribución de * en el directorio `channels/h323`. El canal H.323 sólo actúa como un *gateway* H.323, y no como *gatekeeper*.

Como con todos los canales de *, el archivo de configuración de H.323, llamado `h323.conf`, se sitúa en `/etc/asterisk/`. Un ejemplo del contenido de este archivo se puede ver a continuación.

```
[general]

port = 1720
bindaddr = 0.0.0.0
context=from-h323

;Codecs
disallow=all
allow=gsm
allow=ulaw

;Configuración del GK
gatekeeper=DISCOVER
AllowGKRouted=yes

[300]
type=h323
e164=7091174
context=from-h323

[301]
type=h323
prefix=1234
context=from-h323
```

Puede verse que dentro de la sección `[general]`, los parámetros definidos son muy similares a los de otros canales, a excepción de aquellos que tienen que ver con el *gatekeeper* H.323.

Los posibles valores que puede tomar el parámetro `gatekeeper` son: `DISCOVER`, que indica que se encuentre la dirección IP del *gatekeeper* usando multicast; `DISABLE`, que deshabilita el uso de un *gatekeeper*; y `<IP address>` o `<host name>`, que permiten configurar explícitamente la dirección del *gatekeeper*.

En cuanto al parámetro `AllowGKRouted`, éste define si se aceptarán llamadas enrutadas a través de un *gatekeeper* o no. Por defecto este parámetro está fijado en `yes` pero puede deshabilitarse para tener un control más fino sobre qué tipo de usuarios puede aceptar *.

Finalmente, los parámetros `e164` y `prefix` sólo tienen sentido si se tiene un *gatekeeper* en el sistema, ya que éste será el encargado de asociar todas las llamadas de o hacia el número o prefijo definidos, con la dirección de red y el identificador TSAP correspondiente. En el ejemplo, el número E.164 `7110974` se asocia con la etiqueta `300`, y el prefijo `1234` se asocia con la etiqueta `301`. Internamente, * trabajará con las etiquetas, pero hacia afuera, los clientes H.323 serán identificados por el número E.164 o por el prefijo.

3.1.4. Otros protocolos

Como se dijo anteriormente, * es una PBX extremadamente flexible. Es así que además de los tipos de canales antes descritos, también acepta otros tipos de canales como:

- ISDN (*Integrated Services Digital Network*)
- MGCP (*Media Gateway Control Protocol*)
- SCCP (*Skinny*[2])
- VoFR (*Voice over Frame Relay*)
- Bluetooth[3]

Estos protocolos no se analizarán en la presente documentación ya que no se consideraron parte de nuestro estudio.

3.2. Dialplan

El *dialplan* es el elemento central del sistema. Para entender el funcionamiento de * y configurarlo exitosamente, el usuario debe entender profundamente cómo funciona el *dialplan*. Es a través de éste que el sistema puede enrutar las llamadas en curso, a partir de un listado de instrucciones o pasos que * deberá seguir y que se ejecutan a partir de dígitos recibidos desde un canal o aplicación. La programación del *dialplan* es altamente flexible y permite desarrollar una lógica compleja para el tratamiento de las llamadas, simplemente mediante unas cuantas sentencias o instrucciones.

El *dialplan* se define en el archivo `extensions.conf`, que se ubica usualmente en `/etc/asterisk/` como la gran mayoría de los archivos de configuración del sistema[4]. Este archivo, y por lo tanto el formato general del *dialplan*, se encuentra formado por cuatro elementos fundamentales: *contextos*, *extensiones*, *prioridades* y *aplicaciones*[5].

[2] Protocolo propietario de Cisco utilizado por defecto por los teléfonos IP Cisco y por el Cisco *Call Manager*

[3] Protocolo de interconexión para redes PAN (*Personal Area Networks*); IEEE 802.15.1.

[4] Esto no es mandatorio y puede modificarse, pero en ese directorio se ubican los archivos de configuración tras una instalación estándar de *.

[5] Las aplicaciones serán descritas en detalle en la Sección 3.4, haciendo hincapié en su importancia como elemento de extensión de funcionalidades para el usuario

Los contextos juegan un papel estructural dentro del *dialplan*. Una de sus principales funcionalidades es definir el alcance para, por ejemplo, poder limitar el acceso de ciertos usuarios a servicios pagos como pueden ser las llamadas internacionales. Toda llamada manejada por * proviene de algún contexto y llega a algún contexto, y las instrucciones definidas en dicho contexto serán quienes determinen qué se hará con dicha llamada.

Los contextos dividen el archivo de configuración en secciones. Se define un contexto a partir de una etiqueta escrita entre paréntesis rectos. Por ejemplo, un contexto para manejar el acceso a llamadas internacionales podría definirse así: `[internacional]`.

Bajo este encabezado, todas las instrucciones que se definan pertenecerán a dicho contexto hasta que se defina uno nuevo. Además * permite incluir uno o más contextos previamente definidos dentro de otro. La aplicación más importante para esta funcionalidad es la de organizar en secciones la lógica de manejo de llamadas. De esta forma se podrá limitar o permitir el acceso a diferentes servicios de la PBX. La sintaxis es simplemente `include =>` seguido por el nombre de contexto a incluir. Por ejemplo:

```
[internacional]

include => local
```

Al incluir el contexto `local` dentro del contexto `internacional` se le da el acceso a llamadas locales a aquellas extensiones que puedan realizar llamadas internacionales. Observar que en principio, aquellas extensiones definidas en el contexto local no tendrían acceso a realizar llamadas internacionales.

Dentro de cada contexto se definen una o más extensiones. Trabajando con *, una extensión es una etiqueta que identifica una dirección dentro de un contexto en el *dialplan*. Las extensiones definen cómo es el flujo de llamadas. Dentro del archivo `extensions.conf`, las extensiones se definen mediante una sentencia especial, `exten =>`.

Esta sentencia debe completarse con el nombre de la extensión, una coma, la prioridad, otra coma y finalmente la aplicación a ejecutarse en el canal. Por más que a continuación veremos el detalle sobre prioridades y aplicaciones, un primer ejemplo de la sintaxis completa de una extensión podría ser:

```
exten => 100,1,Dial(SIP/100)
```

De esta sentencia se desprende que si el usuario llega a la extensión 100 del contexto dado, la primera acción tomada por * (prioridad 1) será llamar al canal número 100 de SIP.

Como se observaba del ejemplo anterior, las extensiones comúnmente se forman por secuencias de dígitos. * brinda la posibilidad de poder reconocer patrones de extensiones para el manejo del flujo de llamadas dentro del *dialplan*. Para habilitar este reconocimiento de patrones, se debe comenzar con un _ seguido por un conjunto de caracteres especiales que el sistema sabrá interpretar, y que son los siguientes:

- **X**: representa cualquier dígito del 0 al 9.
- **Z**: representa cualquier dígito del 1 al 9.
- **N**: representa cualquier dígito del 2 al 9.
- **[14-69]**: representa los dígitos definidos dentro de los paréntesis pudiéndose definir rangos con `-`. En el ejemplo serían los dígitos 1, 4, 5, 6 y 9.
- **.**: actúa como *wildcard* y coincide con cualquier patrón.

A modo de ejemplo, si se quisiera definir una condición para que en las extensiones del 1000 al 1999 se reproduzca la música de espera, en el contexto adecuado debería agregarse la siguiente línea:

```
exten => _1XXX,1,MusicOnHold()
```

Más aún, existe un conjunto de extensiones especiales o reservadas por * que pueden ser de gran utilidad a la hora de crear un *dialplan*.

- **s**: La extensión *start* es a la cual pasa en primera instancia el flujo de ejecución dentro de un contexto en caso de no tener información para ejecutar alguna extensión específica.
- **i**: La extensión *invalid* es a la que pasa el flujo de ejecución en caso de que se haya intentado acceder a una extensión no valida para dicho contexto.
- **t**: La extensión *timeout* se utiliza para continuar la ejecución en el caso de aplicaciones donde se espera la respuesta del usuario, y ésta no es recibida luego de cierto tiempo definido previamente.
- **h**: La extensión *hangup*, en caso de estar definida, es a la cual se envían las llamadas una vez que se ha detectado que alguien cortó.

El tercer elemento son las prioridades, que pueden definirse como la secuencia de pasos ordenados en la ejecución de las extensiones. Para cada prioridad, se ejecuta una aplicación. En el caso típico las prioridades para cada extensión comienzan en 1 y se van incrementando de a uno dirigiendo así el flujo de ejecución de aplicaciones de la extensión. Un caso en que esto no se cumple es cuando para una prioridad `n` se ejecuta la aplicación `Dial` (aplicación que permite llamar a una determinada extensión), y la extensión llamada está ocupada. En

ese caso, la prioridad pasa a valer `n+101`, en lugar de `n+1` como ocurre en el resto de los casos.

A continuación se muestra un ejemplo del uso de prioridades.

```
ignorepat => 9

;Salida a teléfono fijo

exten => _9NXXXXXX,1,Dial(Zap/3/${EXTEN:1},20,t)
exten => _9NXXXXXX,2,Playback(unavailable)
exten => _9NXXXXXX,3,Hangup
exten => _9NXXXXXX,102,Playback(busy)
exten => _9NXXXXXX,103,Hangup
```

El ejemplo ilustra la definición de una extensión para acceder a una línea externa y realizar llamadas locales en Montevideo. Para ello se define un patrón con un 9 inicial para tomar la línea y luego una máscara de 7 dígitos para establecer la coincidencia con el número discado. Cuando de una extensión perteneciente al mismo contexto se digita una secuencia de números coherente con la descrita antes, la primera acción es discar por la interfaz Zaptel 3 a dicho número, quitando el prefijo 9 (luego se explicará en detalle cómo esto es implementado). En caso de que no contesten luego de 20 segundos, la ejecución pasa a la siguiente prioridad, donde se ejecuta la aplicación `Playback` que simplemente reproducirá el archivo de audio `unavailable.gsm`[6]. Luego de completarse la reproducción se cierra el canal mediante la aplicación `Hangup`. Por otro lado, en caso de que el número llamado esté ocupado, el flujo de ejecución salta a la prioridad 102 (pues $102 = 1 + 101$), donde se ejecuta el archivo de audio `busy.gsm`, y luego se cierra el canal.

Otro detalle interesante que surge del ejemplo es que, mediante la máscara `NXXXXXX`, se permite discar cualquier número de 7 dígitos que no comience con 0 o 1. El número se le pasa como parámetro a la aplicación `Dial` a través de la **variable** `${EXTEN}`. Ésta se trata de una variable especial reservada de *, donde se guarda el número de extensión en el que se encuentra una llamada. El agregado de `:1` indica que no debe tenerse en cuenta el primer dígito de la secuencia; es así que se puede ignorar el dígito 9.

Las variables son otro elemento interesante que puede utilizarse en el *dialplan* para reducir la escritura de texto repetitivo, agregar claridad o lógica adicional como en el caso visto anteriormente. Las variables se referencian con la sintaxis `${NOMBRE_DE_VAR}` y existen tres tipos diferentes de variables:

- **Globales**, que se definen dentro del contexto especial `[globals]` y tienen alcance sobre todas las extensiones de todos los contextos en el *dialplan*.
- **De canal**, que tienen alcance únicamente sobre la llamada en curso.

[6] `gsm` es el formato de codificación de audio estándar manejado por * para todos los sonidos de la PBX.

- **De entorno**, que permiten acceder a variables de entorno Linux desde *.

Por ejemplo, y siguiendo con el hilo del ejemplo previo, se podría definir una variable global para denotar la interfaz FXO Zaptel de salida para llamadas locales a la PSTN. Es decir:

```
[globals]

LINEA_URBANA=Zap/3

[local]

ignorepat => 9

;Salida a telefono fijo

exten => _9NXXXXXX,1,Dial(${LINEA_URBANA}/${EXTEN:1},20,t)
exten => _9NXXXXXX,2,Playback(unavailable)
exten => _9NXXXXXX,3,Hangup
exten => _9NXXXXXX,102,Playback(busy)
exten => _9NXXXXXX,103,Hangup
```

Por último, se tienen las aplicaciones, que se cargan como módulos dinámicos en * y permiten realizar diversas acciones sobre los canales de voz. En general las aplicaciones aceptan un conjunto de parámetros que especifican lo que el usuario desea que la aplicación realice. Los usuarios de * pueden desarrollar sus propias aplicaciones para extender la funcionalidad del sistema y adaptar la PBX a sus necesidades, como se verá más adelante.

Algunas de las aplicaciones más importantes disponibles con la instalación estándar de * son atender, discar y cortar un canal, reproducir un archivo de audio en un canal o simplemente la música de espera. También se puede enviar el flujo de ejecución a otro contexto y/o extensión, levantar o escribir información de la base de datos interna de * o llamar al sistema de correo de voz para verificar si nos han dejado algún mensaje. La lista de aplicaciones es, por supuesto, mucho más extensa y pueden verse todas las aplicaciones disponibles mediante el comando `show applications` de la línea de comando de * (CLI).

3.3. CLI

La interfaz de línea de comandos de * es una de las herramientas más potentes para conocer el estado actual de la PBX. Para acceder a la misma, basta con agregar el argumento opcional `-c` (modo consola) al iniciar una nueva instancia de *, o la opción `-r` para reconectarse a una instancia existente, que se encuentre corriendo como demonio.

La CLI de * brinda una gran cantidad de comandos para llevar a cabo tareas administrativas, cada uno de ellos con opciones y la posibilidad de autocompletarse mediante el tabulador. El

comando `help` despliega una lista con descripción de todos los comandos que pueden ejecutarse. `help <comando>` despliega información sobre un comando particular.

A continuación se presenta un listado para nada exhaustivo de los comandos disponibles, con la idea de presentar qué tipo de funcionalidad provee la CLI. Para un listado completo de los comandos disponibles consultar `http://voip-info.org/wiki-Asterisk+CLI`.

3.3.1. Comandos generales

Comando	Descripción
`show applications`	Despliega la lista de aplicaciones registradas
`show dialplan`	Muestra el dialplan
`show channels`	Muestra información de los canales
`show voicemail users`	Despliega el listado de casillas definidas
`show conferences`	Muestra el estado de las conferencias
`show parkedcalls`	Despliega el listado de llamadas aparcadas
`add extension`	Agrega una nueva extensión a cierto contexto
`load`	Carga cierto módulo dinámico por nombre

Tabla 3.1: Comandos generales disponibles en * CLI.

3.3.2. Administración de servidor

Comando	Descripción
`restart now`	Reinicia * inmediatamente
`restart when convenient`	Reinicia * tras la finalización de todas las llamadas
`reload`	Carga nuevamente los archivos de configuración
`stop now`	Cierra * inmediatamente
`stop when convenient`	Cierra * tras la finalización de todas las llamadas
`show modules`	Despliega los módulos e información de los mismos
`show version`	Muestra la versión de *

Tabla 3.2: Comandos para la administración de *.

3.3.3. Comandos del canal SIP

Comando	Descripción
`sip debug`	Habilita el *debugging* del canal SIP
`sip no debug`	Deshabilita el *debugging* del canal SIP
`sip reload`	Carga nuevamente el archivo de configuración de SIP
`sip show channels`	Lista los canales SIP activos
`sip show peers`	Lista los clientes SIP registrados a *
`sip show users`	Lista los usuarios SIP definidos en la configuración

Tabla 3.3: Comandos del canal SIP de *.

3.4. Aplicaciones

En las secciones previas, se introdujeron las aplicaciones de * como aquellos módulos que permiten realizar diversas operaciones sobre los canales de voz. Ahora se pretende describirlas haciendo énfasis en la API de aplicaciones, buscando mostrar cómo a través de la misma * se convierte en un potente *framework* para el desarrollo de servicios de telefonía.

Una de las principales ventajas de * como PBX Open Source es que el usuario puede desarrollar sus propias extensiones (o simplemente modificaciones a lo existente) de acuerdo a los requerimientos impuestos por su sistema. Las aplicaciones, a través de una API que define claramente la interacción con el núcleo de *, generan un marco ideal para poder llevar a cabo este tipo de extensiones.

Todas las aplicaciones que se desee utilizar en *, ya sea en el *dialplan* o para alguna otra finalidad puntual, deberán cargarse en * como módulos dinámicos cada vez que se inicializa el servidor. Mediante las funciones `load <aplicación>` o `unload <aplicación>` de la CLI se puede cargar o dar de baja una aplicación en cualquier momento, durante la ejecución de *, siempre y cuando la misma no esté siendo utilizada en dicho instante.

En el Apéndice A se muestra la estructura que debe tener el código para ser una aplicación válida de *. Este archivo fuente se distribuye en las versiones de * bajo el nombre `app_skel.c`.

`load_module` es la función que lleva a cabo el registro inicial de la aplicación cuando arranca la PBX, así como los sucesivos registros que fueran invocados a través del comando `load <aplicación>`. Para que * tenga acceso a dicha función, la biblioteca dinámica (archivo .so) generada luego de la compilación de la aplicación deberá estar en el directorio `/usr/lib/asterisk/modules`.

A través del archivo de configuración `modules.conf` existe la posibilidad de indicarle a * que cargue o no dicho módulo al inicio. Si para una aplicación no se especifica nada, será cargada por defecto.

La implementación de la función `load_module` se muestra a continuación:

```
int load_module(void) {
    return ast_register_application(app, app_exec, synopsis, descrip);
}
```

Se observa que no es más que un *wrapper* de la función `ast_register_application`, implementada en `pbx.c`. Un esquema de los parámetros necesarios para el registro de la aplicación se muestra en la figura 3.1

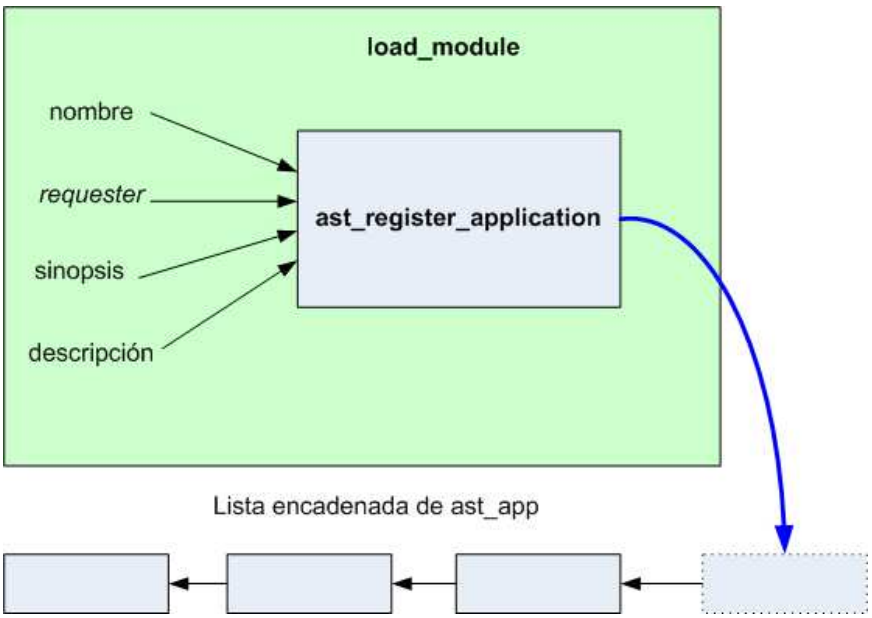


Figura 3.1: Registro de una aplicación en *.

Los parámetros que deben ser especificados para toda aplicación son:

- `app:` *string* con el nombre de la aplicación.
- `app_exec:` puntero a la función que inicia la ejecución de la aplicación (también denominada *requester*). Esta función es la interfaz con la PBX. El nombre usual para estas funciones es `nombre_de_aplicación_exec` (por ejemplo para la aplicación `Dial` es `dial_exec`) pero puede definirse cualquier otro.
- `synopsis:` *string* con un breve identificador de la aplicación. Mediante el comando `show applications` se despliega el listado completo de las aplicaciones cargadas en * con sus respectivas sinopsis.
- `description:` *string* con una descripción de la funcionalidad de la aplicación, parámetros requeridos y forma de ejecutarla. Esta información se despliega mediante el comando `show application <nombre_de_aplicacion>` del CLI.

Al realizar la llamada a la función `ast_register_application` con los parámetros mencionados, se agrega un nuevo elemento a la lista encadenada de estructuras `ast_app` que mantiene los descriptores y *requesters* de todas las aplicaciones cargadas al sistema. Dicha estructura se muestra a continuación:

```
/* ast_app: An application */
struct ast_app {
    char name[AST_MAX_APP];         /* Name of the application */
    int (*execute)(struct ast_channel *chan, void *data);  /* Requester */
    char *synopsis;             /* Synopsis text for 'show applications' */
    char *description;          /* Description for 'show application <name>' */
    struct ast_app *next;           /* Next app in list */
```

Es importante mencionar que la función registrada como *requester* debe aceptar como parámetros de entrada un canal y un `void`.

3.4.1. La aplicación Hola Mundo

Como práctica en lo que refiere al desarrollo de aplicaciones para *, se implementó la sencilla aplicación `app_printtext` que despliega un mensaje en consola. Dicho mensaje puede definirse en el archivo de configuración `printtext.conf` o de lo contrario se despliega "Hola Mundo" por defecto.

Primero que nada, se carga la configuración del archivo, a través de la función `load_config`, que llama a la función de * `ast_load`.

`ast_load` devuelve una estructura del tipo `ast_config` (llamada `cfg`) con la información del archivo de configuración *parseada*, o un puntero a `NULL` si no existe el archivo de configuración. En este caso, el mensaje a desplegar toma el valor por defecto (`deftexten`). Esto se muestra en el código que sigue:

```
cfg = ast_load(PRTEXT_CONFIG);
if (!cfg) {
    ast_log(LOG_WARNING,
    "No se pudo cargar la conf para la aplicación PrintText: %s\n", PRTEXT_CONFIG);
    strncpy(text, deftexten, sizeof(text) - 1);
    return -1;
};
```

Si el archivo de configuración existe, primero se chequea si está definido el contexto `[general]`. Para ello se utiliza la función `ast_variable_browse`, que devuelve 0 si no encuentra el texto que se le pasa como parámetro, en el archivo que también se le pasa como parámetro. Si el contexto `[general]` no está definido, también se carga el valor de texto por defecto. Esto es implementado por el siguiente código:

```
var = ast_variable_browse(cfg, "general");
if (!var) {
    /* nada configurado */
    ast_log(LOG_WARNING,"No existe general.\n");
```

```
    strncpy(text, deftexten, sizeof(text) - 1);
    return -1;
}
```

Luego se buscan las variables que definen el texto a desplegar y el idioma (que determina el idioma del texto por defecto si no se quiere especificar un texto dado), y se almacenan en las variables correspondientes del código de la aplicación (`text` y `lang` respectivamente). Esto se hace mediante el siguiente código:

```
  /* Se carga en lang el valor del lenguaje */

  if (!(s=ast_variable_retrieve(cfg, "general", "lang"))) {
        ast_log(LOG_WARNING,"Lenguaje no especificado. Tomo lenguaje por defecto.\n");
  } else {
        ast_log(LOG_WARNING,"Lenguaje especificado.\n");
        strncpy(lang, s, sizeof(lang) - 1);
  }

  /* Se carga en text el mensaje a desplegar */

  if (!(s=ast_variable_retrieve(cfg, "general", "text"))) {
        ast_log(LOG_WARNING,"Texto no especificado. Tomo texto por defecto.\n");

        /* Si el lenguaje es español... */
        if (!strcmp(lang,"es")) {
        ast_log(LOG_WARNING,"Lang=%s\n",lang);

        /* Se carga en text el texto en español por defecto */
        strncpy(text, deftextes, sizeof(text) - 1);

        /* Si el lenguaje no es español... */
        } else {
        ast_log(LOG_WARNING,"Lenguaje por defecto:%s\n",lang);

        /* Se carga en text el texto en inglés por defecto */
        strncpy(text, deftexten, sizeof(text) - 1);
        }
  } else {
        ast_log(LOG_WARNING,"Texto especificado.\n");

        /* Se carga en text el texto especificado en el archivo de configuración */
        strncpy(text, s, sizeof(text) - 1);
  }
```

Luego de finalizada la carga del archivo de configuración, se procede al registro de la aplicación. El mismo se hace mediante la sentencia:

```
res = ast_register_application(app,printtext_exec,synopsis,descrip);
```

Como se ve a partir de esta sentencia, el *requester* de la aplicación es `printtext_exec`. A continuación se muestra el código que es ejecutado al invocar esta función:

```
static int printtext_exec(struct ast_channel *chan, void *data) {
    ast_log(LOG_WARNING, "El texto a desplegar es:\n %s\n", text);
```

```
    return 0;
}
```

Como se había mencionado anteriormente, el valor de `text` se carga desde el archivo de configuración correspondiente, y de no estar definido toma el valor por defecto.

Notar también que los dos parámetros que se le pasan al *requester* son `struct ast_channel *chan` y `void *data`, lo cual es mandatorio en *, por más que en este caso no sean usados por la función, dada su simplicidad.

El código completo de esta aplicación puede leerse en el Apéndice B.

4. EL CANAL SIP DE *

Este capítulo presenta un estudio más profundo y de bajo nivel de la implementación del protocolo SIP dentro de *.

Primero se realizará una descripción del canal SIP mencionando las principales funcionalidades que brinda ya sea como servidor o como cliente. Luego se introduce el archivo de configuración `sip.conf`, necesario para establecer comunicaciones a través del canal SIP. Finalmente se hace un seguimiento del flujo de ejecución de * (a nivel de código C) una vez que se detecta que un usuario desea realizar una llamada utilizando SIP.

4.1. Descripción general del canal SIP

El módulo para el canal SIP permite que * establezca comunicaciones VoIP con otros agentes de usuario SIP en la red. Gracias a este módulo * puede actuar como:

- **cliente SIP**, para lo cual debe registrarse en un servidor SIP para poder realizar y recibir llamadas desde el servidor. Todas las llamadas entrantes serán manejadas de acuerdo al *dialplan* para derivarse a alguna extensión existente.
- **servidor SIP**, donde clientes SIP dentro de la red puedan registrarse para establecer sesiones SIP con * y realizar llamadas. Actuando como servidor, todo lo que refiere a mensajes SIP entre usuarios registrados a * pasa y es procesado por el servidor. En cuanto al audio, según la configuración elegida existen ambas posibilidades: que el flujo RTP pase por * o que se intercambie directamente entre los puntos finales. En general, tras establecer la llamada, * envía mensajes conocidos como REINVITE para redireccionar el flujo RTP directamente entre las puntas. Hay ocasiones donde esto no es posible y el audio se ve forzado a encaminarse vía *. Un par de ejemplos de esta situación serían terminales SIP que no manejan REINVITEs o directamente cuando los terminales no acuerdan un codec común y * debe permanecer en el medio para hacer la traducción.
- ***gateway***, brindando conversión de señalización y medios con otras tecnologías como la PSTN u otros protocolos VoIP como IAX2 o H.323.

Al hablar de servidores SIP es usual suponer que uno se refiere a un *proxy* SIP. Sin embargo * no actúa como servidor *proxy* sino más bien como un B2B (*Back To Back*), es decir, como un par cliente/servidor, servidor/cliente. Un *proxy* se encarga momentáneamente del control de llamadas entre terminales SIP pero nunca actúa como punto final; además, no necesariamente

mantiene el estado de las llamadas una vez que las mismas se establecen.

Por el contrario, el módulo SIP de * se ha desarrollado de tal forma que al manejar el control de una llamada entre dos terminales A y B, * resulta ser el UA destino de la llamada de A para luego establecer una nueva llamada a B. En la figura 4.1 se muestra el flujo de mensajes SIP para establecer una llamada entre dos *softphones* registrados en * (A utilizando un X-Lite, B un Linphone).

Se observa cómo el servidor responde como agente destino al INVITE del llamante y envía uno nuevo al cliente llamado estableciendo las dos sesiones SIP para una llamada, a diferencia del *proxy* que sólo reenvía el INVITE que recibe. Por esta razón * permanece siempre en el medio controlando y manteniendo el estado de la llamada. En este caso donde no se envían REINVITEs para redireccionar el flujo de medios, los canales se interconectan mediante lo que se conoce como *puenteo*.

Esta estructura de manejo de canales separados $A \Leftrightarrow *$ y $* \Leftrightarrow B$ presenta muchas ventajas pero también tiene sus aspectos negativos. La gran ventaja de esta implementación es que favorece el funcionamiento de * como *gateway*. En principio, los canales establecidos entre cada terminal y * pueden ser de cualquier tipo, siempre y cuando sean implementados respetando el API de canales. Esta especificación de una estructura común para los canales permite la abstracción de la función de *puenteo*, y como consecuencia la interconexión de dos tecnologías cualesquiera para las que se haya diseñado un módulo de comunicación. Sin embargo, esta arquitectura complica la implementación de algunas funciones simples como puede ser la transferencia de llamadas. SIP resuelve las transferencias mediante un mensaje especial REFER, pero es claro que en implementaciones con * se requiere funcionalidad y complejidad adicional para hacer y deshacer las interconexiones entre canales.

Volviendo a la situación donde interesa redireccionar el flujo RTP entre las puntas, veremos cómo se implementan los REINVITEs mediante el estudio de la transacción de inicio de llamada que se muestra en la figura 4.2. Cabe mencionar que el REINVITE no es un tipo de mensaje especial, sino que es simplemente un segundo INVITE con nueva información RTP. En este escenario, el *softphone* X-Lite con dirección IP 192.168.0.101 inicia una llamada hacia el Linphone con IP 192.168.0.15. Ambos clientes se encuentran registrados al *, cuya dirección IP 192.168.0.10. En primera instancia, la llamada se establece normalmente como dos canales hacia el *, tal cual se vio anteriormente. El canal entre el X-Lite y * se inicia mediante el INVITE con autentificación que aparece como mensaje número 4 en el *callflow*. Un fragmento del *payload* SDP en dicho mensaje es:

```
Session Description Protocol Version (v): 0
Owner/Creator,SessionId (o): 102 18754868 18754898 IN IP4 192.168.0.101
Session Name (s): X-Lite
Connection Information (c): IN IP4 192.168.0.101
Media Description, name and address (m): audio 8000 RTP/AVP 0 98 97 101
```

Figura 4.1: Establecimiento de una llamada en *.

La información relevante es que el X-Lite informa que su RTP se encontrará en la dirección IP 192.168.0.101 puerto 8000. * le responde con su información SDP en el mensaje 200 OK que se muestra con el número 14.

```
Session Description Protocol Version (v): 0
Owner/Creator, Session Id (o): root 17383 17383 IN IP4 192.168.0.10
Session Name (s): session
Connection Information (c): IN IP4 192.168.0.10
Media Description, name and address (m): audio 15906 RTP/AVP 0 8 4 3
```

Durante este proceso, se negocia de manera similar el canal de salida donde * le envía el INVITE con SDP al Linphone (mensaje número 6) y éste le responde con su información SDP en el mensaje 200 OK que se muestra con el número 12. El fragmento SDP relevante contenido en esta respuesta es el siguiente:

```
Session Description Protocol Version (v): 0
Owner/Creator, Session Id (o): gmateos 123456 654321 IN IP4 192.168.0.15
Session Name (s): A conversation
Connection Information (c): IN IP4 192.168.0.15
Media Description, name and address (m): audio 10500 RTP/AVP 0 8 3 110 101
```

En este caso, el Linphone indica que su RTP se encontrará en la dirección IP 192.168.0.15 puerto 10500.

En este punto, * conoce la información SDP de ambos softphones y se encuentra en condiciones de enviar los REINVITEs para redirigir el flujo de medios. El primero de estos REINVITEs se envía al Linphone en el mensaje número 15. * mantiene el `Call-ID` que estaba utilizando para el canal de salida, de manera que el Linphone reconozca que se trata de dicha sesión. El SDP de este mensaje incluye la información RTP del X-Lite:

```
Session Description Protocol Version (v): 0
Owner/Creator, Session Id (o): root 17383 17384 IN IP4 192.168.0.101
Session Name (s): session
Connection Information (c): IN IP4 192.168.0.101
Media Description, name and address (m): audio 8000 RTP/AVP 0 110 97 101
```

Un punto importante a tener en cuenta es que solamente se redirecciona el flujo RTP, la sesión SIP para este canal sigue establecida entre el Linphone y *. Es decir, si el Linphone quisiera finalizar la llamada, le enviará un mensaje BYE a * y no al X-Lite.

Para completar el proceso, * envía un segundo REINVITE al X-Lite (mensaje número 20) con la información RTP del Linphone en el *payload* SDP y el `Call-ID` correspondiente al canal entre el X-Lite y *:

```
Session Description Protocol Version (v): 0
Owner/Creator, Session Id (o): root 17383 17384 IN IP4 192.168.0.15
Session Name (s): session
Connection Information (c): IN IP4 192.168.0.15
Media Description, name and address (m): audio 10050 RTP/AVP 0 101
```

4.2. Configuración de canales SIP

Para poder realizar llamadas SIP mediante *, es necesario definir como mínimo los archivos de configuración `sip.conf` y `extensions.conf`[1]. Como se vio anteriormente, el archivo `extensions.conf` es común a todos los canales y define la lógica de manejo de llamadas dentro de la PBX. Por el contrario, `sip.conf` es un archivo específico del protocolo SIP y permite definir parámetros generales del canal, los diferentes clientes que podrán registrarse en el servidor o los servidores a los cuales * se registrará en caso de funcionar como cliente SIP.

[1] Ambos se encuentran usualmente en el directorio `/etc/asterisk/`

Figura 4.2: Establecimiento de una llamada en *, con mensajes REINVITE.

Los parámetros generales de configuración del canal se definen bajo el encabezado `[general]`, en la parte superior del archivo `sip.conf`. Algunas variables relevantes con sus posibles valores se listan a continuación.

- `allow=[codec]`. Habilita globalmente la utilización del codec definido. Los codecs soportados por * son `ulaw`, `alaw`, `g723.1`, `g726`, `g729`, `gsm`, `ilbc` y `speex`.
- `disallow=all`. Deshabilita globalmente todos los codecs. En los esquemas generalmente utilizados, primero se deshabilitan todos para luego habilitar solamente aquellos requeridos por la aplicación.
- `bindaddr=0.0.0.0`. Especifica las direcciones IP en las cuales * escuchará por posibles conexiones. La dirección 0.0.0.0 implica todas las direcciones definidas en la máquina.
- `context=[nombre_del_contexto]`. Indica el contexto por defecto al cual irán las llamadas SIP entrantes en el *dialplan*.
- `externip=x.x.x.x`. Dirección IP pública a colocar en los mensajes SIP en caso de que el servidor se encuentre detrás de un NAT.
- `localnet=x.x.x.x/y.y.y.y`. Dirección y máscara para la red privada detrás del NAT.
- `port=5060`. Puerto en el cual se debe escuchar por posibles conexiones. Usualmente se utiliza el puerto SIP por defecto, 5060.
- `register=>usuario:clave@host:puerto/extensión`. Se define únicamente en el caso de que * trabaje como cliente y especifica los datos necesarios para registrarse en el servidor `host`.
- `videosupport=yes|no`. Habilita o deshabilita el soporte para sesiones de video.
- `realm=[realm]`. Define el dominio (en inglés, *realm*) para los registros con autentificación. Por defecto utiliza el valor `asterisk`.

Luego de la sección general de configuración, se deben incluir los parámetros específicos de cada uno de los clientes que podrán realizar llamadas entrantes a * (*users*), salientes de * (*peers*) o ambas (*friends*). En el caso de que * trabaje como cliente SIP registrándose a un servidor para sus llamadas salientes, se deberá configurar el *peer* correspondiente al servidor en esta sección.

Las variables de configuración de cada cliente se definen tras un encabezado de la forma `[nombre_del_cliente]`. Alguna de las variables relevantes que pueden configurarse son:

- `type=user|peer|friend`. Define si el cliente realiza llamadas entrantes, salientes o ambas.
- `defaultip=x.x.x.x`. Dirección IP estática para el cliente.

- `host=dynamic`. En caso de que el cliente pueda registrarse desde diferentes direcciones IP, deberá definirse esta variable en sustitución de `defaultip`.
- `context=[nombre_del_contexto]`. Contexto en el *dialplan* al cual irán las llamadas entrantes de este cliente.
- `dtmfmode=inband|info|rfc2833` . Define la forma en que el cliente maneja la señalización de tonos multifrecuentes. En el caso de *Inband*, los tonos son tratados como audio común, siendo generados por los terminales; debido a esto, son distorsionados por aquellos codecs que utilizan compresión (todos excepto G.711). En el caso de *Info*, los DTMFs se transmiten fuera de banda a través de INFO, método específico de SIP. Finalmente, en el caso de RFC 2833 [18], se define un tipo especial de contenido RTP para la transmisión de los tonos.
- `username=[usuario]`. En el caso de clientes que llaman a *, este nombre de usuario debe ser el utilizado en INVITEs hasta registrarse. En el caso de que * como servidor requiera de INVITEs autenticados, este deberá ser el nombre de usuario que el cliente envía para el registro. De manera similar, para el caso de * como cliente conectado a un servidor SIP, se usará este valor para las transacciones a efectuar con dicho servidor (llamadas entrantes o salientes). Por lo tanto, este valor debe ser igual al definido en la variable global `register`. Ésta puede verse como una redundancia, pero la implementación actual de * así lo requiere.
- `secret=[pass]`. En el caso de que el cliente se registre con * para realizar llamadas, deberá utilizar este password para ello. En el caso de que * funcione como cliente, se aplican los mismos comentarios que en el ítem anterior.
- `port=5060`. Puerto utilizado por el cliente.
- `nat=yes|no`. Modifica el funcionamiento de * para el caso en que el cliente se encuentre detrás de un NAT. Notar que esto no se aplica en el caso de que * esté detrás de un NAT y el cliente en el exterior. Esto se verá con más detalle en la sección 5.3.
- `rtptimeout=[limite]`. Define el tiempo máximo antes de cortar la llamada si no se detectan paquetes RTP para la sesión. Se deshabilita si la llamada se pone en espera.
- `musiconhold=[clase_de_musica]`. Define la clase de música de espera a reproducir para llamadas de este cliente SIP.

Observar que muchos de los parámetros globales (como `port`, `allow`, `disallow`, etc.) pueden re-definirse para cada cliente en particular, sobreescribiendo el valor global. Por ejemplo, se puede tener deshabilitado el codec G.711 μ a nivel global, pero un cliente específico requiere su uso; en ese caso, se habilita el mismo a nivel del cliente mencionado.

A continuación se muestra un ejemplo sencillo de la configuración en `sip.conf`, necesaria para que el cliente `X-Lite` con nombre de usuario `100` y clave `clave_X`, pueda registrarse en * para realizar y recibir llamadas.

```
[general]

port = 5060             ;Puerto al cual conectarse (el puerto SIP
                        ;es 5060)
bindaddr = 0.0.0.0      ;Direccion a la cual conectarse(todas las
                        ;de la maquina)
disallow=all            ;Deshabilita todos los codecs
allow=gsm               ;Habilita distintos codecs uno a uno
allow=alaw
allow=ulaw
context = bogon-calls   ;Contexto al que se envia a los usuarios
                        ;no identificados, definido en extensions.conf.


[X-Lite]

type=friend             ;Este usuario puede realizar y recibir
                        ;llamadas
username=100            ;Nombre de usuario para registrarse al
                        ;servidor
secret=clave_X          ;Clave del usuario
host=dynamic            ;Este usuario no se conecta siempre desde
                        ;la misma direccion IP
context=from-sip        ;Las llamadas de este usuario pertenecen
                        ;a este contexto
```

Observar que este archivo tiene el formato estándar de todos los archivos de configuración manejados por * (diferentes secciones entre [ ] y variables definidas dentro de cada uno), ya que el *parseo* de los mismos se hace de manera estándar y es implementado en una única función.

Finalmente se muestra un nuevo ejemplo de configuración sencillo para el caso en que * funciona como cliente registrándose en el servidor SIP de un proveedor. La idea de este ejemplo es presentar la configuración mínima para establecer este tipo de conexión, por más que no se hayan realizado pruebas de este tipo en el transcurso del proyecto. La plataforma de estudio ha sido, en general, un * como servidor al cual se registran diversos clientes SIP dentro de la LAN o posiblemente fuera de ella a través de Internet y por lo tanto no se ha abordado el funcionamiento de * como cliente.

```
[general]

disallow=all
allow=gsm
allow=alaw
allow=ulaw
context = bogon-calls
register => 2345:password@miproveedorSIP.com/1234
```

```
; Registra la extensión 1234 de * como el usuario
; 2345 en el servidor del proveedor

[miproveedorSIP]

type=peer
secret=password
username=2345
host=servidor.miproveedorSIP.com
fromuser=2345
fromdomain=midominio.com
```

Como se dijo anteriormente, si bien la información en el campo `register` siempre debe ser la misma que la que aparece en la definición del *peer* asociado al servidor (`[miproveedorSIP]`), esto es necesario debido a la actual implementación de *.

4.3. Manejo de llamada SIP en *

Todos los canales de * se implementan mediante un módulo dinámico que es cargado por * en su inicialización. A nivel de código fuente, toda la funcionalidad del módulo SIP se encuentra en el archivo `chan_sip.c`. Allí se definen las principales estructuras de datos y funciones que dan al sistema la funcionalidad deseada como cliente o servidor SIP. Es importante mencionar que se utilizan bibliotecas estándar de C pero también muchas funciones del propio *, ya que la arquitectura del sistema en su globalidad busca abstraer la funcionalidad del canal, de las demás funciones centrales de la PBX. La funcionalidad del núcleo del sistema es la que se obtiene de dichas bibliotecas propias de *, y es la que permite establecer el vínculo con el canal SIP.

Dentro de esta sección se describirá el flujo de ejecución de * una vez que se detecta una llamada entrante desde un cliente SIP. Se buscará introducir y describir brevemente la funcionalidad de los principales métodos que atraviesa dicho flujo de ejecución para procesar la llamada.

4.3.1. Inicialización

La función `load_module` es la que permite cargar el módulo dinámico que implementa el canal SIP una vez que se inicializa *. Todos los módulos que implementan canales así como aplicaciones para brindar diversas funcionalidades en el *dialplan* deben incluir esta función para cargarse en el sistema. Dentro de esta función se llevan a cabo los siguientes pasos:

1. se llama a `reload_config`, que se encarga de *parsear* el archivo de configuración `sip.conf`. Como se mencionaba en la sección previa, esto se hace de manera estándar mediante una función de * llamada `ast_load`, que toma como parámetro el nombre del

archivo de configuración (`sip.conf`) y devuelve una estructura del tipo `ast_config` donde se guarda toda la información definida en dicho archivo.

2. a partir de los datos almacenados en esta estructura se inicializan variables globales con lo cargado de la configuración del usuario en la sección `[general]`.

3. se cargan las listas encadenadas de estructuras de *users* y *peers* hasta completar el contenido del archivo de configuración. Los *users* se guardan en estructuras `sip_user`, análogamente los *peers* en `sip_peer`[2].

4. se toma la información de dirección IP y puerto SIP (definidos en las variables `bindaddr` y `port` de la sección `[general]` de `sip.conf`) para crear mediante la función `socket`, el *socket* UDP donde se escuchará por mensajes SIP destinados a *.

`ast_channel_register_ex` es la función que dentro de `load_module` efectivamente registra el canal SIP. De manera similar a como se vio para las aplicaciones en la sección 3.4, para el registro de un canal se deben especificar los siguientes parámetros:

- el nombre del canal (es decir "SIP", utilizado por ej. en la aplicación `Dial(SIP/xxx)`).
- una descripción textual.
- los codecs soportados.
- el *requester*, puntero a la función que asigna canales SIP. Dicha función es `sip_request`, y es uno de los principales puntos de conexión entre la PBX y el módulo del canal.

En el final de `load_module`, luego de procesar todo lo necesario para el registro del canal, se llama a la función `restart_monitor` que inicializa la funcionalidad del canal al llamar a `do_monitor`.
`do_monitor` es el *thread* que monitorea el arribo de posibles llamadas. Llama a `ast_io_add` que designa la función a ser llamada en caso de que el monitor detecte un *socket* UDP. Dicha función es `sip_sock_read` y en caso de un evento se le pasa el flujo de ejecución para procesar el *socket* recibido.

4.3.2. Captura de mensajes SIP en el socket

En `sip_sock_read` se define una estructura de datos `sip_request` para almacenar el paquete SIP recibido y un puntero a `sip_pvt`, otra estructura de datos fundamental que mantiene la información privada de cada llamada[3]. Mediante el llamado a la función `recvfrom` se copia el paquete SIP en bruto dentro del campo `data` de la estructura `sip_request` y luego mediante el llamado a `parse` se termina de procesar el mensaje reconociendo los diferentes encabezados SIP y líneas del posible *payload* SDP. Esta información se carga en los

[2] En el Apéndice C se da el esquema detallado de estas estructuras de datos.
[3] Ídem nota anterior.

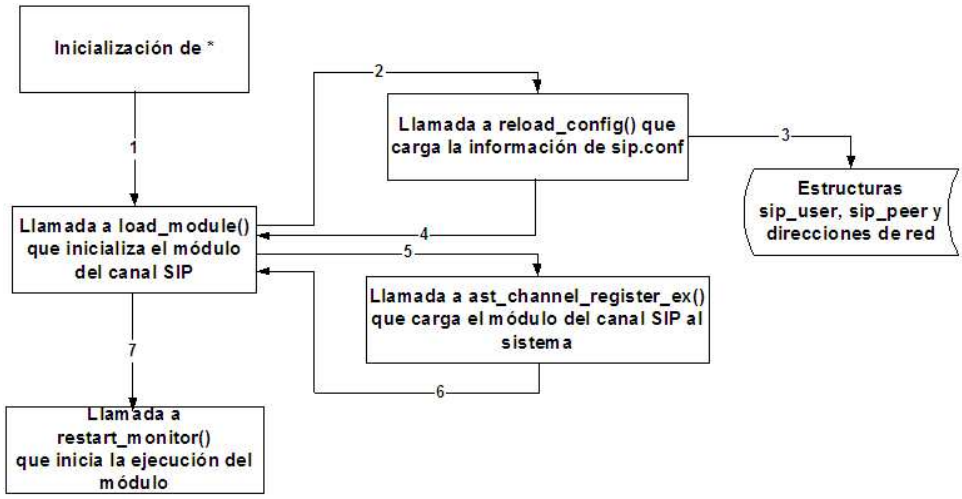


Figura 4.3: Inicialización del módulo dinámico que implementa el canal SIP.

arreglos `header` y `line` respectivamente del `sip_request`. También se guarda la dirección IP del *socket* en una variable. Finalmente se chequea que `sip_request` tenga como mínimo dos encabezados, y si no sale de la función indicando error.

Si el anterior chequeo es positivo se llama a la función `find_call` pasándole como parámetros el mensaje SIP recibido y la dirección IP de origen. Esta función busca si el mensaje SIP se relaciona con alguna llamada SIP existente o en caso contrario crea un nueva. La búsqueda se realiza por el `call_id` del mensaje (que unívocamente determina una llamada) recorriendo `iflist`, la lista encadenada de estructuras `sip_pvt`. Si encuentra una llamada con igual `call-id` que el mensaje SIP recibido, retorna devolviendo la estructura `sip_pvt` correspondiente a la llamada. De lo contrario se trata de una nueva llamada y debe asignarle una nueva estructura, para lo cual se llama a `sip_alloc`.

4.3.3. *Inicialización de la estructura privada de la llamada*

La funcionalidad de `sip_alloc` es esencialmente la de asignación de memoria para la estructura `sip_pvt` asociada a la nueva llamada, y la inicialización de sus campos a los valores por defecto.

Por ejemplo, se inicializan todos sus campos de direcciones IP y se le asignan valores a los campos de dirección de origen (que se pasaba como parámetro de entrada) y dirección del * que se obtuvo tras el `reload_config` inicial. A continuación se inicializa la sesión RTP y el correspondiente RTCP. Esto se lleva a cabo mediante el llamado a la función `ast_rtp_new_with_bindaddr` de `rtp.c`, archivo fuente donde se implementa todo el soporte para dicho protocolo en *. Luego se llenan los campos para la estructura NAT y demás parámetros globales relevantes, con valores por defecto.

Finalmente se agrega la nueva estructura `sip_pvt` a `iflist` retornando el flujo de ejecución a `find_call` y luego a `sipsock_read`. Luego, se llama a la función `handle_request`, quien se encarga de procesar todos los distintos tipos de mensajes de solicitud SIP que arriban a la PBX desde los clientes registrados.

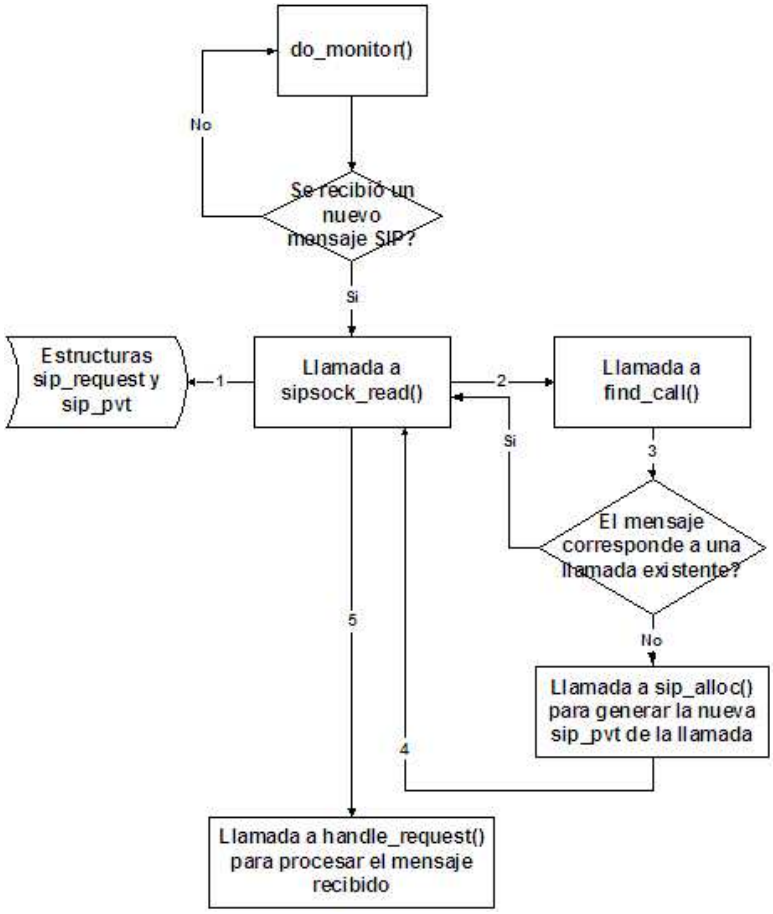


Figura 4.4: Lectura de socket SIP y procesamiento preliminar de mensajes recibidos

4.3.4. *Procesamiento de solicitudes*

`handle_request` recibe como parámetros la `sip_pvt` privada de la llamada, los datos del mensaje recibido en la estructura `sip_request` y la dirección IP del origen. En primer lugar, se compara la primera línea del paquete recibido contra el valor `SIP/2.0`. Si la comparación es positiva, no se trata de una solicitud sino de una respuesta perteneciente a una transacción SIP existente. En ese caso, hace unos chequeos sobre el paquete, en particular verifica que el número de secuencia recibido sea mayor al que se tiene almacenado como número de secuencia de salida en la estructura `sip_pvt`. Si todo es correcto, se llama a la función auxiliar

`extract_uri` que extrae el URI del *Contact* y lo guarda en el campo correspondiente de la estructura privada de la llamada. Finalmente, se le pasa el flujo de ejecución a la función `handle_response`, quien se encarga de manejar la respuesta recibida.

Volviendo al chequeo inicial, si el mensaje recibido se trata de una solicitud, primero compara el número de secuencia que tiene almacenado en la estructura `sip_pvt` contra el recibido en el campo `Cseq` del mensaje. Si el paquete es viejo, se sale de la función indicando error. Si el número de secuencia es correcto se actualizan los campos `tag` y `lastmsg` de la estructura privada, con la información del campo `From` (sólo el número después de `tag=`) y el tipo de solicitud recibida, respectivamente.

Tras estas verificaciones iniciales se pasa al cuerpo principal de la función donde se chequea de qué tipo de solicitud se trata, y se procesa de manera acorde para continuar correctamente la transacción SIP con el cliente que ha enviado la solicitud. Los posibles tipos de solicitud manejados por * son: `OPTIONS`, `INVITE`, `REFER`, `CANCEL`, `BYE`, `MESSAGE`, `SUSCRIBE`, `INFO`, `NOTIFY`, `REGISTER` y `ACK`.

En lo que refiere al procesamiento de una llamada entrante, las dos solicitudes de interés son `INVITE` y `ACK` por lo cual serán ellas las que se verán a continuación.

4.3.5. El INVITE: inicio de una llamada

Si la solicitud recibida corresponde a un `INVITE`, se procede de acuerdo a los siguientes pasos:

1. se verifica que no sea una llamada a nosotros mismos.
2. se actualiza el campo `pendinginvite` de la estructura `sip_pvt` con el valor extraído del número de secuencia del `INVITE`, para luego saber que existe una transacción de inicio de llamada pendiente.
3. se guarda una copia del paquete recibido en el campo `initreq`.
4. se hace un llamado a la función `check_via` que se encarga de extraer la información de *host* del campo `Via` y luego realizar una consulta DNS para obtener la dirección IP correspondiente al *host*. Si la consulta es exitosa, la dirección se guarda en la estructura `sip_pvt`.
5. se verifica si se trata del primer `INVITE` recibido para dicha transacción y en caso afirmativo se verifica si el usuario está configurado en *, mediante el llamado a `check_user`. Esta función se encarga de extraer la información del campo `From` para cargar el valor de la extensión en la estructura `sip_pvt` y para luego realizar búsquedas en las listas encadenadas de *users* y *peers*. La función `find_user` es la que recorre todos los *users* configurados en busca de alguno con igual nombre de usuario que el extraído del `From`.
6. si encuentra alguno, devuelve una estructura `sip_user` para que puedan actualizarse los campos relevantes en la estructura privada de la llamada, con los valores configurados en `sip.conf` para el *user*.

7. si no encuentra un *user* se llama a `find_peer` que hace una búsqueda en la lista `peerl` por la dirección IP del cliente que envió el `INVITE`. En caso de encontrarlo, también se devuelve una estructura `sip_peer` para que se pueda actualizar el `sip_pvt` con los datos cargados en `sip.conf`.

8. en caso de no haber coincidencia luego de ambas búsquedas, se retorna a `handle_request` y se llama a `transmit_response` para enviar una respuesta del tipo `403 Forbidden`.

9. si alguna de las búsquedas fue exitosa, en `handle_request` se continúa con el procesamiento del *payload* SDP recibido en el `INVITE`. Para ello se llama a `process_sdp` que se encarga de extraer la información de codecs y RTP del cliente.

10. se chequea que efectivamente haya *payload* RTP y luego se extraen una a una las líneas de datos mediante la función `get_sdp` que toma como parámetros la estructura `sip_request` y un identificador de la línea de interés.

11. se actualiza el dato del puerto RTP para audio del cliente y con la información de su lista de codecs soportados se negocia la compatibilidad con *.

12. en caso afirmativo se actualiza el campo `jointcapability` en la `sip_pvt` y se retorna a `handle_request`. En caso de incompatibilidad se retorna y se llama a `transmit_response` para enviar una respuesta del tipo `488 Not acceptable here`.

13. como verificación de que pueda realizarse la llamada en ese momento, se incrementa el contador de llamadas activas mediante un llamado a la función `update_user_counter` con uno de sus argumentos igualados a `INC_IN_USE`.

14. se compara con el máximo de llamadas permitidas y en caso de error retorna indicando que no es posible procesar la llamada.

15. si todo es correcto se llama a la función `get_destination` que se encarga de extraer del mensaje recibido, la URI destino para actualizar la extensión y el dominio en la estructura `sip_pvt`. Si no se encuentra el destino, se envía una respuesta del tipo `404 Not Found`.

16. mediante la información de extensión y contexto cargadas en dicha estructura se buscará una coincidencia en el *dialplan* para conectar esta llamada entrante a través de la ejecución de cierta aplicación según sea la configuración del `extensions.conf`.

17. se llama a `extract_uri` que extrae la información de la URI origen para actualizar la estructura privada y finalmente llama a `build_contact` que arma el encabezado `Contact` para los mensajes que * envíe, con la extensión y la dirección IP del servidor.

4.3.6. Solicitud del canal de entrada

Siendo este mensaje el primer `INVITE`, se pasa a una parte muy importante que es la asignación de un nuevo canal para la llamada. Esto en los hechos se logra mediante el *seteo* del

campo `owner` de la estructura `sip_pvt` a un canal cuya información se guarda en una estructura del tipo `ast_chan`. Para ello se llama a la función `sip_new` que toma como parámetros la estructura `sip_pvt` y el estado actual de llamada que se codifica como `AST_STATE_DOWN`. Esta función devuelve la estructura `ast_chan` asignada a la llamada.

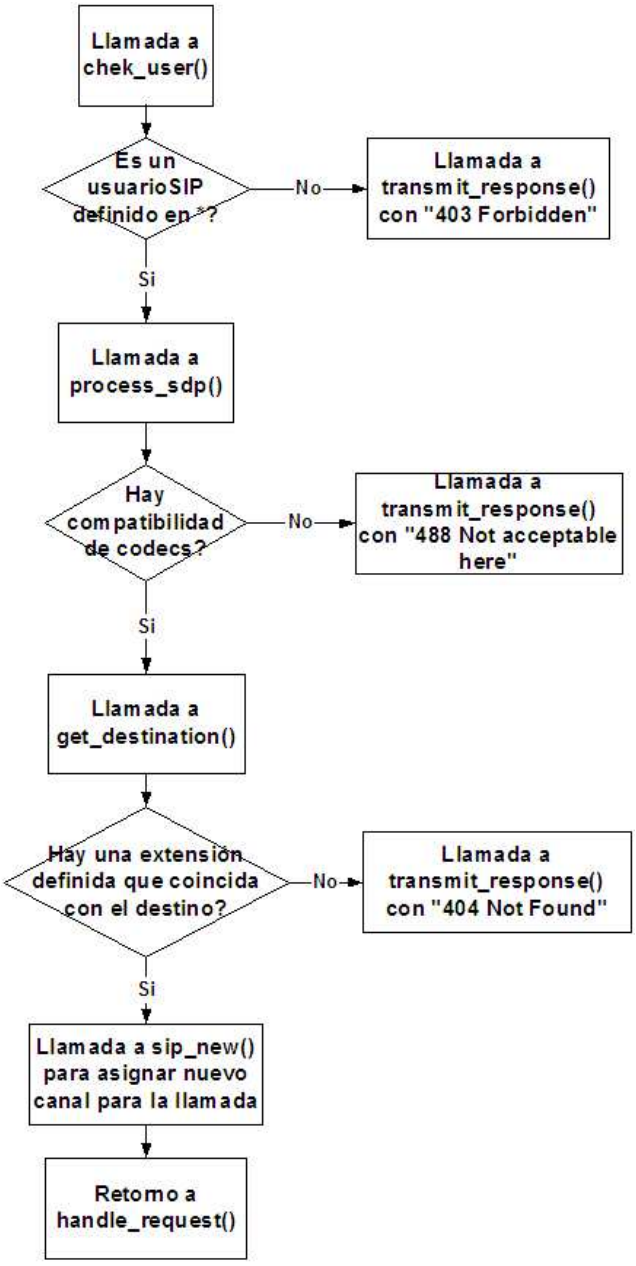


Figura 4.5: Procesamiento del primer INVITE.

En sip_new primero que nada se crea el nuevo canal. Luego se asignan diversos campos del mismo a partir de lo almacenado en la estructura sip_pvt. Notar que todo aquello particular a la tecnología SIP es lo que se almacena en esta última estructura privada. Es en sip_new que se da la interfaz con el núcleo de * ya que de alguna forma, los datos necesarios para la estructura ast_chan son quienes plantean los requerimientos genéricos para que cualquier tecnología implementada como módulo dinámico pueda comunicarse a través de *. En particular se define el nombre de todas las funciones relevantes (definidas en chan_sip.c) para llevar a cabo y cortar llamadas, y para recibir y enviar audio RTP de forma que el núcleo de * pueda llamarlas. Luego del mapeo de unas cuantas variables de la estructura privada a la estructura del canal, el canal también guarda el estado actual de la llamada, llamando para ello a ast_set_state con parámetro AST_STATE_DOWN. Finalmente con el nuevo canal para la llamada ya inicializado se vuelve a handle_request.

Primero se guarda la posible ruta al origen especificada en el campo Record-Route del INVITE. Para ello se llama a la función build_route que extrae esta información del encabezado y la guarda en la estructura privada utilizando estructuras del tipo sip_route. Ya se ha completado lo que sería el tratamiento del primer INVITE y el flujo de ejecución ha alcanzado la parte donde se procesa el INVITE si el estado de la llamada es más avanzado que AST_STATE_DOWN. En primer lugar se chequea que efectivamente haya un canal asignado para la llamada y luego si el estado es:

- AST_STATE_DOWN: se inicia la transmisión de la respuesta 100 Trying mediante un llamado a transmit_response. Luego se compara la extensión a la cual se quiere llamar contra la devuelta por la función ast_pickup_exten y en caso afirmativo se le pasa finalmente el flujo de ejecución al núcleo del sistema mediante el llamado a la función ast_pbx_start implementada en pbx.c con el canal asociado a la llamada como parámetro. Esta función se encarga de crear un nuevo thread del *dialplan* para luego llamar a la función ast_pbx_run que se encarga de la búsqueda de aplicaciones definidas para ir ejecutándolas y conectar el canal.
- AST_STATE_RING: se manda trasmitir la respuesta 100 Trying mediante un llamado a transmit_response.
- AST_STATE_RINGING: se manda trasmitir la respuesta 180 Ringing mediante un llamado a transmit_response.
- AST_STATE_UP: se manda transmitir la respuesta 200 OK mediante un llamado a la función transmit_response_with_sdp. En este caso ya se ha negociado la llamada con el otro extremo y ambas partes pueden comunicarse.

4.3.7. El ACK: final de la transacción de inicio de llamada

Para finalizar, había quedado pendiente el estudio de cómo se manejaba la recepción de una solicitud ACK, que era relevante en la transacción de inicio de llamada ya que es de esperar un mensaje de este tipo después de que * envíe una respuesta 200 OK. En este caso se compara el

número de secuencia del paquete recibido (extraído al comienzo de `handle_request`) contra el campo `pendinginvite` de la estructura. Si la comparación es positiva, se trata de un `ACK` correspondiente a la transacción de inicio que estaba activa y se *resetea* dicho campo en la estructura `sip_pvt`. Finalmente se llama a la función `__sip_ack` que reconoce la recepción del paquete y detiene las posibles retransmisiones que estuvieran programadas (por ejemplo de una respuesta `200 OK` para la cual no se haya recibido un `ACK`).

4.4. La otra mitad del proceso: el canal de salida

Para completar el estudio del manejo de una llamada SIP en *, se presentará un seguimiento desde el momento en que la PBX con un canal de entrada, conociendo la extensión destino, se encuentra al recorrer el *dialplan* con que debe ejecutar la aplicación `Dial(SIP/XXX)`. Se verá cómo la PBX solicita un nuevo canal SIP de salida y se completa la llamada. Este seguimiento será mucho menos detallado que el de la sección previa, solamente se buscará indicar las funciones de mayor relevancia y sus propósitos.

Suponiendo que un cliente SIP ha iniciado una llamada y luego de todo el proceso descrito en la sección previa, se le pasa un canal a la PBX con la extensión destino, las funciones de núcleo de * recorren el *dialplan* para ver cómo manejarla. Dado que para dicha extensión destino, con primera prioridad se encuentra definida la aplicación `Dial(SIP/XXX)`, la PBX pasará a ejecutar el contenido de la aplicación. El archivo fuente de la aplicación `Dial` es `app_dial.c` y se encuentra junto al resto de las aplicaciones en `/usr/src/asterisk/apps`.

Todas las aplicaciones que se desarrollen para * y que se desee ejecutar desde el *dialplan*, deberán ser registradas indicando la función que será llamada al momento de requerir dicha ejecución. Para el caso de la aplicación `Dial`, la sentencia de registro se muestra a continuación.

```
res = ast_register_application(app, dial_exec, synopsis, descrip);
```

Como se mencionó anteriormente en la sección 3.4, el segundo parámetro de la función `ast_register_application` es el que define cuál será la función llamada (en este caso, `dial_exec`). Luego, la PBX llamará en primer lugar a la función `dial_exec` pasándole el canal de entrada y los argumentos definidos en el *dialplan*.

4.4.1. Solicitud del canal de salida

En la función `dial_exec` luego de realizar diversos procesamientos iniciales sobre los argumentos que recibe, solicita un canal de salida que se establecerá entre * y el *peer* destino de la llamada. Para ello realiza una llamada a la función `ast_request` pasándole la tecnología (SIP), los codecs soportados por el canal de entrada y el usuario destino como parámetros. `ast_request` es una función genérica de * para solicitar canales de cualquier tecnología. Se encuentra implementada en el archivo fuente `channel.c`. * maneja una estructura de datos llamada `chanlist` que almacena una descripción de cada una de las tecnologías que se registran como canales. En particular contiene un campo fundamental denominado `requester`

donde se guarda el nombre específico de la función a llamar para solicitar el canal de dicha tecnología. La información de todos los canales se almacena en una lista encadenada llamada `backends`.

En la sección previa se estudió el registro del canal SIP en la PBX al ejecutar la función `ast_register_ex` y donde se pasaba como argumentos en nombre de la tecnología, el *requester* y los codecs soportados. Es en la función `ast_register_ex` donde se agrega un nuevo elemento `chanlist` a `backends` con los datos del canal particular.

Para el caso del canal SIP el *requester* es la función `sip_request` que como es de esperarse retorna una estructura del tipo `ast_chan`. Por lo tanto, volviendo a la función `ast_request`, ésta simplemente recorre la lista encadenada de tecnologías de canales hasta encontrar la tecnología para la cual se desea solicitar un canal (SIP en este caso). Una vez que encuentra la estructura `chan_list` correspondiente a la tecnología, hace un llamado a su *requester*.

Por lo tanto el flujo de ejecución pasa a la función `sip_request` en `chan_sip.c`. Esta función esencialmente lo que hace es asignar una nueva estructura `sip_pvt` para la llamada saliente (mediante a un llamado a `sip_alloc`) e inicializar algunos de sus campos. Finalmente hace un llamado a la función `sip_new` que ya se ha estudiado en la sección previa, y cuyo objetivo es asignar un nuevo canal con la estructura privada creada anteriormente. Por lo tanto, este nuevo canal con información específica para la tecnología SIP es el que se retorna a la función `ast_request` y consecuentemente a `dial_exec` que lo había solicitado.

4.4.2. Inicio de la llamada en el nuevo canal

Nuevamente en `dial_exec`, luego de la asignación de algunos campos del nuevo canal de salida a partir de la información contenida en el canal de entrada, se inicia la llamada al *peer* destino mediante la llamada a `ast_call`. Esta última función es muy similar a `ast_request` en el sentido de que es genérica de * y hace un llamado a la función específica de la tecnología que se encarga de iniciar llamadas en el canal. El nombre de dicha función a llamar se encuentra en la estructura privada almacenada en el canal de salida, y para el caso de SIP es `sip_call`.

En la función `sip_call` se extrae la estructura privada del canal que se pasa como parámetro, y se marca la llamada como saliente. Se actualiza la cantidad de llamadas salientes activas mediante el llamado a `update_user_counter`, y si el máximo no se excede se inicia la transacción SIP con el UA destino mediante un llamado a `transmit_invite`.

Mientras que a nivel de `chan_sip.c` se procesa la transacción de inicio de llamada, se retorna a `dial_exec` haciéndose un llamado a `wait_for_answer` que aguarda a que se establezca la llamada en el canal saliente. Si la llamada es atendida en un tiempo menor al establecido, primero se verifica si los canales pueden compatibilizarse para realizar un *puenteo* y en caso afirmativo se hace un llamado a `ast_bridge_call` con los canales de entrada y salida como parámetros. Esta función es la que se encarga de copiar los paquetes de audio de un canal a

otro en ambos sentidos para permitir la llamada.

En la figura 4.6 se muestra un diagrama que representa el procesamiento del canal de salida descrito en esta sección.

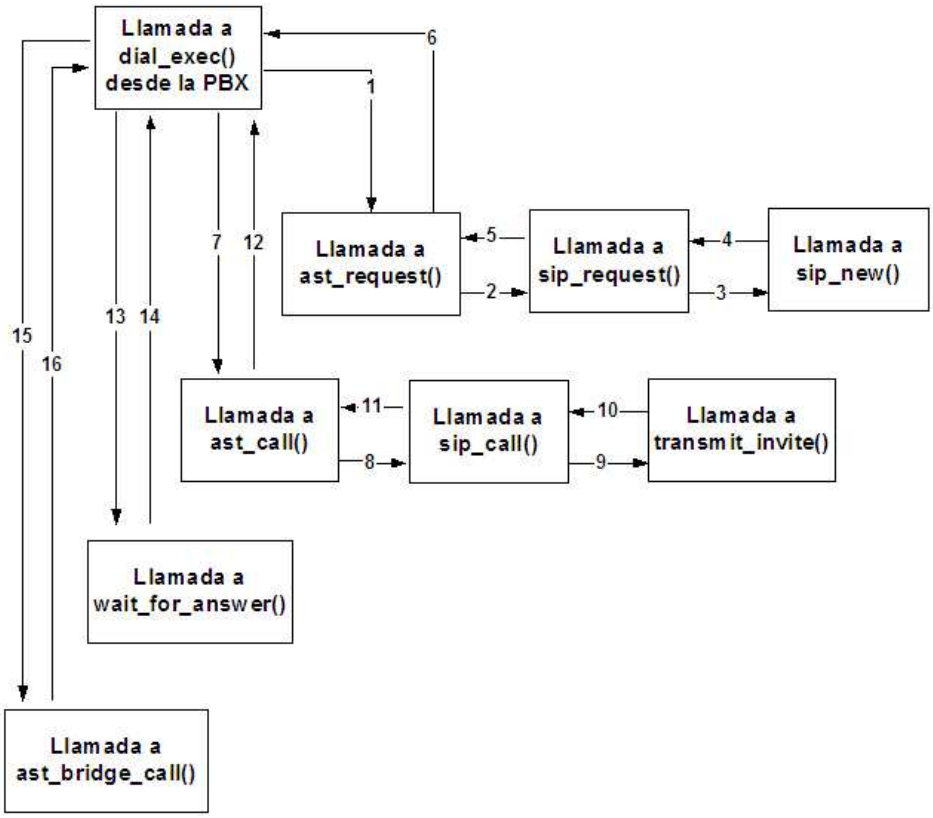


Figura 4.6: Procesamiento del canal de salida.

5. SIP & *: PRIMEROS ACERCAMIENTOS Y ESCENARIOS DE PRUEBA

En este capítulo, se presentan aplicaciones específicas que permitieron adentrarnos desde diversos puntos de vista al estudio del funcionamiento de SIP utilizando *.

Se describen algunas modificaciones sencillas realizadas al código fuente de la PBX, de manera de lograr objetivos puntuales de funcionamiento pero con el objetivo principal de generar caminos de acercamiento hacia el código y el lenguaje de programación C.

En segundo lugar, se introducen los diferentes clientes SIP utilizados en el transcurso del proyecto para la realización de pruebas de comunicación. Se busca presentar sus respectivas funcionalidades y dar ejemplos sencillos de configuración para permitir su interacción con *.

Luego, se presentan los resultados de un estudio detallado realizado para evaluar la funcionalidad de * en lo que refiere a la llamadas SIP en escenarios con NAT.

Por último, se describe el estudio de funcionalidad avanzada brindada por * para la realización de transferencias de llamadas SIP. Se contrastan las posibilidades de implementar las transferencias mediante mensajes SIP (según especifica la IETF en sus recomendaciones), o directamente con funcionalidad de * mediante el *puenteo* de canales.

5.1. Modificaciones en el código

En la etapa inicial de estudio de *, se modificó el código de aplicaciones puntuales, de forma de alcanzar objetivos específicos y sencillos. Dentro de este marco modificamos los archivos `chan_sip.c` para la parametrización de la variable `realm` utilizada en los mensajes de autentificación SIP , y `say_number.c` para la parametrización del idioma en que se dicen los números pregrabados de *.

5.1.1. Parametrización de realm

Nuestro objetivo en esta etapa consistió en parametrizar la variable `realm`, cuyo valor originalmente era constante e igual a “asterisk”, pudiéndose pasar su valor como parámetro en la sección `general` de `sip.conf`.

El procedimiento a seguir fue el siguiente:

- Dentro del archivo `chan_sip.c`, se definió la variable estática `my_realm`:

```
static char my_realm[100];
```

- Se buscó en `chan_sip.c`, la función que carga los datos del archivo de configuración `sip.conf`, a saber, `reload_config`. Se modificó dicha función de forma de que se asigne en la variable `my_realm` el valor del campo `realm`, opcionalmente configurable en la sección `general` de `sip.conf`. Se le asignó por defecto el valor de "asterisk", para mantener la compatibilidad hacia atrás en caso de no configurar un valor diferente.
- Donde aparecía `Digest realm = asterisk` (como argumento de varias funciones), se cambió `asterisk` por `%s\\`, y se agregó la variable `my_realm` a los argumentos.

Las pruebas realizadas para evaluar el funcionamiento y los resultados obtenidos se describen a continuación:

- Se ejecutó `ngrep SIP port 5060` para poder capturar los mensajes SIP en la interfaz. En el campo `System Settings | SIP Proxy | Realm` del softphone X-Lite, se configuró `asterisk`, y no se configuró el campo `realm` en `sip.conf`. Al registrarse el X-Lite, en los paquetes capturados por el ngrep el campo `Digest realm` de los encabezados de autenticación aparece como:

  ```
  Proxy-Authenticate: Digest realm=asterisk,nonce=31d6a7cb
  ```

  y el registro fue exitoso.
- Se cambió el valor de `asterisk` del X-Lite por `pepito.com`, y también se agregó `realm =pepito.com` en la sección `general` de `sip.conf`. Verificamos que nuevamente el registro fue exitoso, y la información en los encabezados de autentificación es coherente.
- Se cambiaron alternativamente cada uno de los campos, y el X-Lite no pudo registrarse.

5.1.2. Parametrización de idioma en say_number.c

El objetivo puntual en esta etapa era permitir seleccionar entre dos idiomas, español e inglés, a partir de la variable `language` en la sección general de `sip.conf`. Esto implica la traducción de mensajes y números (especialmente esto último) de inglés a español, no sólo teniendo en cuenta su pronunciación sino también la lógica con que se arman los números (por ejemplo, cómo se logra decir el número 123 = ciento + veinti + tres).

El procedimiento llevado a cabo se describe a continuación:

- Se estudió el archivo `say.c` ubicado en la carpeta `usr/src/asterisk/`. Dentro de la función `ast_say_number`, se agregó la posibilidad de elegir entre dos lenguajes, a partir de la sentencia:

  ```
  if strcmp(language,"es");
  ```

Si el lenguaje no es español ("es"), se considera lenguaje inglés ("en") por defecto. Luego se prosiguió con las modificaciones necesarias en la lógica de armado de números a reproducir por las diversas aplicaciones.

- Para distinguir entre "cien" y "ciento", se agregó en la parte de la función que chequea si existe o no la centena en el número, la siguiente distinción:

  `if (!num)`
  se reproduce "cien"
  `else`
  se reproduce "ciento"

  (lo que se muestra arriba es simplemente pseudo-código, donde `num` es la variable que almacena a la parte de decenas y unidades del número que queremos decir, por ejemplo si queremos decir 123 entonces `num` = 23).

- Se distinguieron los casos en que se debe decir "veinte" y "veinti". Para ello, se guardaron ambas palabras en archivos de audio distintos; "veinte" se comporta como el "twenty" del inglés, mientras que se debe decir "veinti" en cualquier otro caso en que se diría la decena e "y" (i.e. decir "veinti" es lo mismo que decir "treinta y", por ejemplo).

- Se agregó la reproducción del archivo "y" luego del número de decena. Aquí se tomó en cuenta el hecho de tener que diferenciar el caso de decena = 2 ("veinti").

- Se guardaron los archivos de sonido en español en `/var/lib/asterisk/sounds/es/digits`. Todo el contenido anterior del directorio `/var/lib/asterisk/sounds/digits` fue movido a `/var/lib/asterisk/sounds/en/digits`.

- En el caso de la aplicación Voicemail (Correo de Voz), se hizo la distinción para el caso en que la cantidad de mensajes viejos o nuevos sea uno, ya que en ese caso en lugar de tener que decir "uno", el sistema debe decir "un".

- Para grabar los nuevos archivos de sonido en español, se definió una nueva extensión dentro del *dialplan* y en ella se agregó la aplicación `Record`, eligiendo el formato `gsm` para los archivos de sonido. Para probar la calidad de la grabación, se reprodujeron mediante la aplicación `Playback`.

Para la validación de las modificaciones realizadas, se llevaron a cabo las siguientes pruebas:

- Se invocó al servicio de `Voicemail`, que hace llamadas a la función `ast_say_number` cuando informa la cantidad de mensajes nuevos y viejos en la casilla del usuario. Se probó con diferentes llamadas, observando el comportamiento ante distintas cantidades de mensajes, incluyendo el caso en que se tuviera un solo mensaje nuevo o viejo.

  Simultáneamente, en la consola de * se fueron monitoreando los archivos de audio reproducidos, chequeando que estos fueran los esperados a priori. Durante el desarrollo y validación de las modificaciones al programa, resultó especialmente de ayuda el hecho de poder observar en la consola de * lo que iba pasando, ya que muchas veces para

chequear si se estaba entrando o no a ciertas funciones (o a partes de ellas), se agregaron líneas del tipo:

```
ast_log("AST_WARNING",...)
```

que despliegan mensajes definidos por el programador en la consola.

- Se probó con `language=en`, `language=es` y sin especificar lenguaje, y en todos los casos se obtuvieron los resultados esperados. Cabe mencionar que las modificaciones realizadas para la reproducción en español, no toman en cuenta números mayores que un millón. De todas formas, parece no ser necesario para las aplicaciones en que se usa la función `ast_say_number`.

5.2. Clientes SIP

En la presente sección se presentarán algunos de los clientes SIP utilizados con *, a la hora de realizar las diferentes pruebas de comunicación. En primer lugar consideraremos tres *softphones*, dos de ellos para Windows y el otro para Linux, finalizando luego con el ATA 186 funcionando como *gateway* SIP. Los *softphones* gratuitos disponibles en Internet, resultaron ser una herramienta de gran ayuda para evaluar las funcionalidades de *, ya que la configuración es en general rápida y directa, la interfaz amigable y sus funcionalidades exceden la de varios teléfonos IP sencillos disponibles en el mercado.

5.2.1. X-Lite

X-lite es un softphone gratuito para sistemas Windows (98SE, NT4.0, ME, 2000, XP), Pocket PC 2000/2002 y MAC OS X. Hemos trabajado con la versión 2.0 disponible en Internet en el sitio: `http://www.xten.com`. Entre las principales funcionalidades de este softphone podemos listar:

- Cumple con el estándar SIP (RFC 3261) y DTMF (RFC 2833)
- 3 líneas, información de llamada (caller ID, duración, etc.)
- Posibilidad de registrarse en múltiples *proxies* SIP (hasta 10)
- Múltiples codecs soportados (G711, XPX, iLBC, GSM)
- Soporte para NAT/*Firewall*

En la figura 5.1 puede observarse la interfaz de usuario del X-Lite.

Figura 5.1: Interfaz de usuario del X-Lite

Configuración básica

Buscaremos presentar la configuración básica para que un terminal X-Lite pueda registrarse al * para realizar y recibir llamadas.

Supongamos la siguiente configuración para la extensión 101 (correspondiente al terminal X-Lite) en el archivo de configuración de *, `sip.conf`:

```
[general]

disallow = all
allow = ulaw
allow = alaw
allow = gsm
port = 5060
bindaddr = 0.0.0.0
context = bogon-calls

[101]

type = friend
username = 101
```

```
secret = gonchi
host = dynamic
context = from-sip
mailbox = 101
```

Se han habilitado los codecs G.711 (Ley A y Ley μ) y GSM soportados por el X-Lite. Si la dirección IP de la máquina donde se encuentra corriendo el * es 164.73.38.24 la configuración necesaria en el X-Lite se puede observar en la figura 5.2.

Figura 5.2: Configuración en X-Lite.

A dicho menú se accede en:

```
Menu | System Settings |SIP Proxy | default
```

Otro detalle es que debe deshabilitarse la supresión de silencios para evitar las advertencias por RFC 3389 en la consola del *. Para ello, setear la siguiente opción:

```
Menu | Advanced System Settings | Audio Settings |
Silence Settings | Transmit Silence: Yes
```

5.2.2. *Firefly*

El Firefly es otro *softphone* gratuito para Windows, que puede obtenerse directamente de Internet en `http://www.virbiage.com/download.php`. La interfaz de usuario se muestra en la figura 5.3
Sus principales funcionalidades incluyen:

- Soporte para los protocolos VoIP SIP e IAX2 (propietario de *)

Figura 5.3: Interfaz de usuario del softphone Firefly

- Posibilidad de registrarse en múltiples servidores SIP e IAX (hasta 10)
- Múltiples codecs soportados (G711, G.729, SPEEX, iLBC, GSM)

Es un *softphone* menos completo que el X-Lite, pero al brindar soporte IAX2 ha sido de utilidad para realizar pruebas con este nuevo protocolo. De todas formas, nos enfocaremos en su configuración para realizar llamadas SIP estando registrado al *.

Nuevamente supongamos la siguiente configuración de partida para la extensión 101 (correspondiente al Firefly) en el archivo de configuración de *, `sip.conf`:

```
[general]

disallow = all
allow = ulaw
allow = alaw
allow = gsm
port = 5060
bindaddr = 0.0.0.0
context = bogon-calls

[101]

type = friend
username = 101
```

```
secret = gonchi
host = dynamic
context = from-sip
mailbox = 101
```

Si la dirección IP del * es 200.40.20.2, la configuración necesaria para que el usuario 101 pueda registrarse se muestra en la figura 5.4:

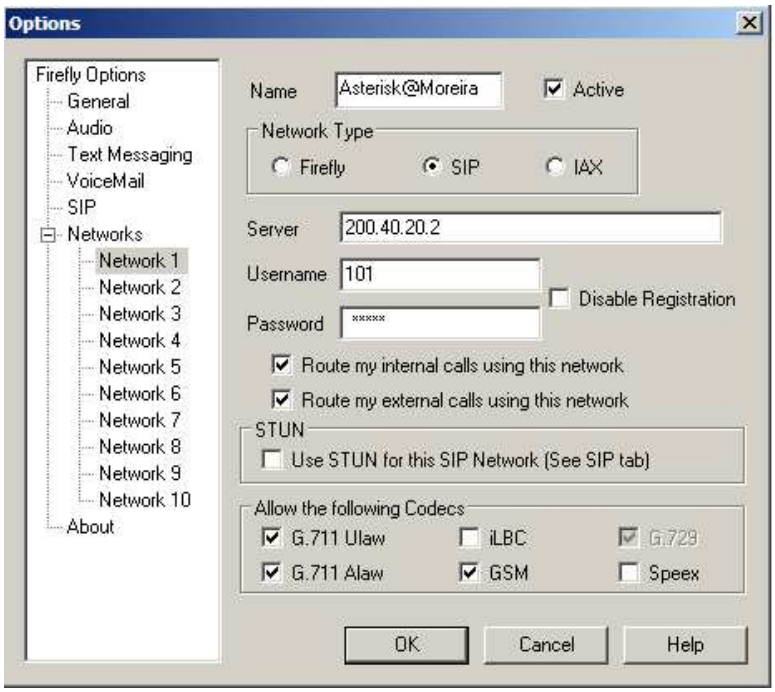


Figura 5.4: Menú de configuración SIP del Firefly

También debería verificarse que la siguiente opción se encuentra elegida, ya que es la soportada por defecto por * para la transmisión de DTMF:

```
Menu | SIP | DTMF Signalling: RTP (RFC 2833)
```

5.2.3. *Linphone*

El Linphone es un *softphone* SIP, Open Source para Linux. Ha sido de fundamental importancia en el proyecto, ya que su soporte para IPv6 ha permitido realizar las pruebas luego de las modificaciones realizadas para agregar soporte IPv6 al canal SIP de *[1]. El software se encuentra disponible en Internet, en `http://simon.morlat.free.fr/download/`. La versión utilizada es 1.0.1, que brinda soporte IPv6. La principal funcionalidad del *softphone* puede resumirse en los siguientes puntos:

[1] El desarrollo de funcionalidad IPv6 para el canal SIP de * se describe en detalle en el Capítulo 6

- Posee interfaz gráfica para el entorno Gnome o puede también ejecutarse en modo consola a través del comando `linphonec`.
- Soporte para SIP de acuerdo al RFC 3261, en redes IPv4 e IPv6.
- Múltiples codecs soportados (G711, G.729, SPEEX, iLBC, GSM).
- Soporte para DTMF (RFC 2833 y ENUM).

La interfaz gráfica del Linphone se muestra en la figura 5.5.

Figura 5.5: Interfaz de usuario del Linphone.

Presentaremos un ejemplo de configuración, para la situación en que el * se encuentra en la dirección IPv6 c006::2. En primer lugar, en el menú de configuración `Network` debe habilitarse la utilización de IPv6. Marcar `Use IPv6 network` como se muestra en la figura 5.6.
Luego hay que configurar los aspectos relativos a * como servidor de registro. Para ello dentro del menú `SIP`, presionar el botón de `Add Proxy/Registrar`. Se despliega el menú de la figura 5.7, donde se debe ingresar la identidad del usuario SIP correspondiente al Linphone con el formato:

```
sip: nombre_de_usuario@[dirección_ipv6_del_*]
```

Es importante seguir esta sintaxis, en particular los paréntesis rectos en la dirección IPv6.[2] Luego hay que ingresar la dirección IPv6 del *, con el formato:

[2] El Linphone no le da muchas opciones al usuario, si el formato no se respeta como se especifica, la aplicación se cuelga.

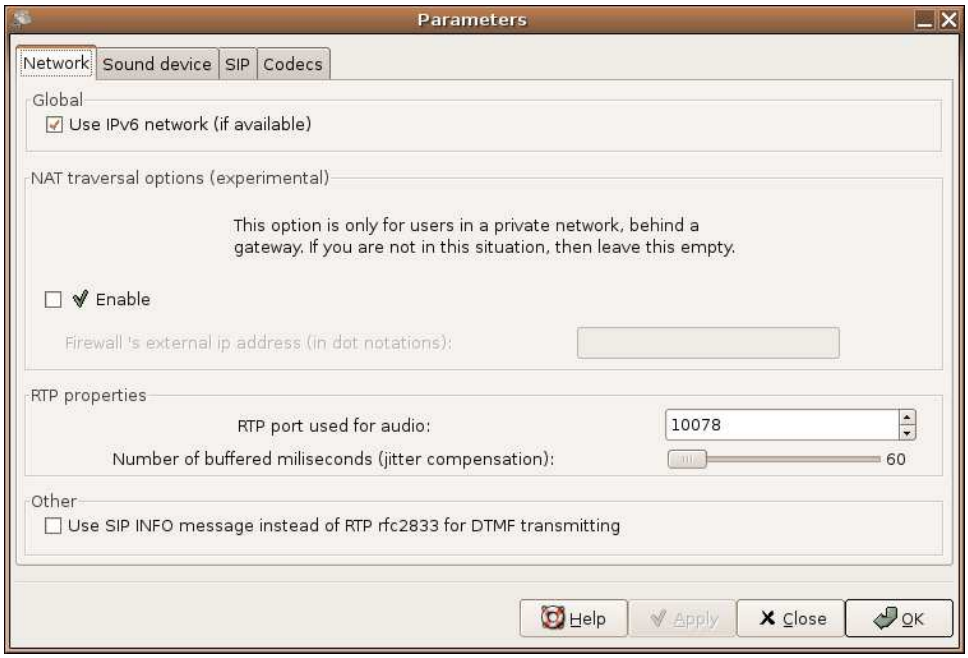


Figura 5.6: Habilitación de IPv6.

```
sip: [dirección_ipv6_del_*]
```

En el contexto de este ejemplo, se supone que el usuario SIP definido en * es el número 100.

Finalmente, a partir de los codecs definidos para el usuario en la configuración en `sip.conf`, se deben habilitar los mismos en el menu `Codecs`. También puede definirse el orden de prioridad, de acuerdo a las necesidades de la aplicación particular.

5.2.4. Cisco ATA 186

El ATA (Analog Telephone Adapter) 186 de Cisco es un *gateway* que permite convertir teléfonos analógicos tradicionales en teléfonos IP. El ATA incluye interfaces FXS para la conexión de hasta dos teléfonos y un puerto Ethernet 10/100BaseTx para la conexión a la red IP. Puede utilizarse con las diferentes tecnologías de VoIP: SIP, H.323, SCCP y MGCP. En cada caso deberá cargarse la imagen de software adecuada y no puede haber más de una cargada a la vez. Un ejemplo de escenario de telefonía IP dentro de una red privada utilizando ATA se muestra en la figura 5.8.

Configuración básica

Al igual que con los *softphones*, presentaremos el procedimiento de configuración básico para que cada una de las interfaces del ATA-186 se registre a * y se puedan realizar llamadas con

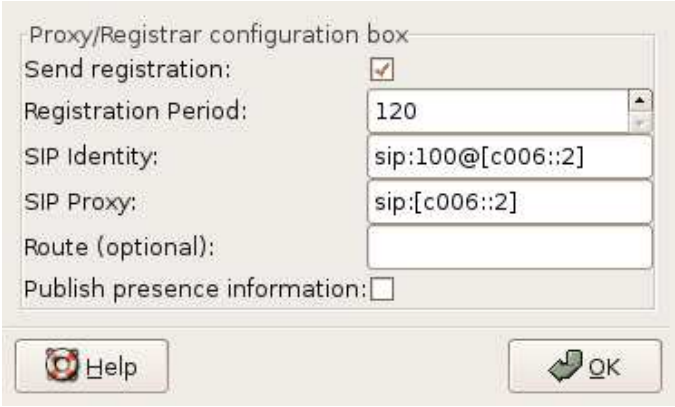


Figura 5.7: Configuración de *.

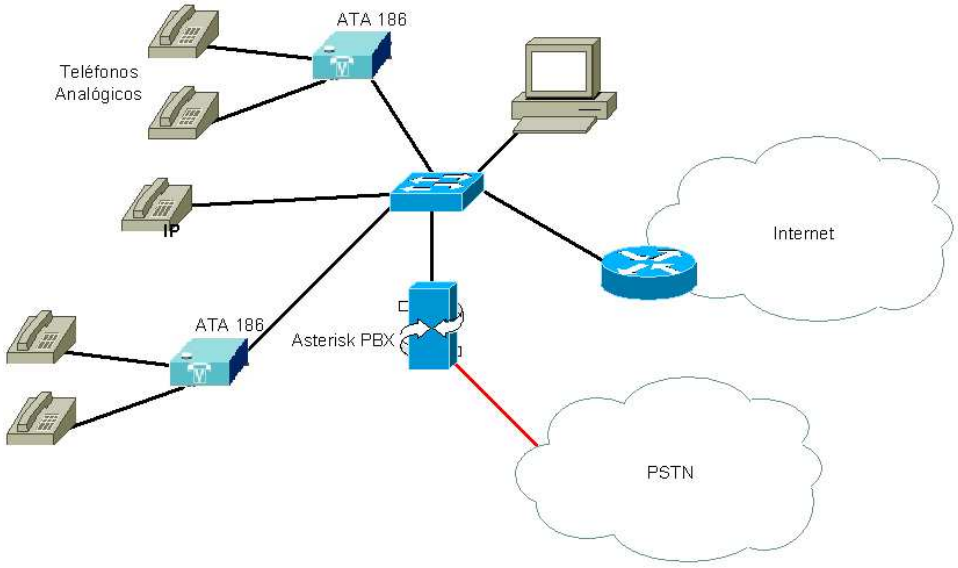


Figura 5.8: Escenario de telefonía IP con ATA 186

señalización SIP. Supondremos que el ATA tiene cargada la imagen de software adecuada para SIP/H-323.

En primera instancia, es conveniente resetear los parámetros de configuración del ATA a sus valores por defecto. Para ello, se debe conectar la alimentación del ATA y un teléfono analógico a la primera de las líneas (identificada como “Phone 1”). Existe una secuencia especial de teclas en el teléfono para resetear el equipo. El procedimiento es el siguiente:

1. Descolgar el microteléfono.
2. Presionar el botón rojo en el ATA.

3. Tras escuchar "Configuration menu, enter menu number…", digitar la siguiente secuencia: `322873738#`.
4. Tras escuchar "To save press star, or press the pound key", presionar: `*`.
5. La luz roja brillará y el ATA arrancará con los parámetros por defecto.

En caso de que el ATA esté conectado a una red con servidor DHCP, éste le asignará una IP tras el reinicio. Para saber cuál es esa IP:

1. Descolgar el microteléfono.
2. Presionar el botón rojo en el ATA.
3. Tras escuchar "Configuration menu, enter menu number…", digitar la siguiente secuencia: `80#`.

En caso de no disponer de un servidor DHCP, o simplemente querer fijar una dirección IP de manera estática, se puede seguir un procedimiento de configuración con secuencias de teclas disponible en el sitio web del fabricante (`www.cisco.com`).

Luego deberán configurarse en * las extensiones para cada una de las líneas del ATA. En este ejemplo supondremos los números de extensión 101 y 102 para las interfaces 1 y 2 respectivamente. El fragmento relevante en `sip.conf` sería:

```
[101]

type = friend
username = 101
secret = gonchi
host = dynamic
dtmfmode = rfc2833
context = from-sip
mailbox = 101

[102]

type = friend
username = 102
secret = mauge
host = dynamic
dtmfmode = rfc2833
context = from-sip
mailbox = 102
```

La configuración es estándar como para cualquier extensión SIP que ya hemos visto. En particular observar la línea `dtmfmode=rfc2833` que define cómo se pasan los tonos entre el

ATA-186 y *. La otra posibilidad es mandar los tonos multifrecuentes en el canal de audio con `dtmfmode=inband`, pero esto no funciona del todo bien en algunos casos.

Luego debe configurarse el ATA para que pueda registrarse al *, de acuerdo a la configuración de extensiones definida arriba. Suponiendo que la dirección IP asignada al ATA es 164.73.38.240, para acceder al menú web de configuración del mismo deberá abrirse el explorador web y acceder a la siguiente página: `http://164.73.38.240/dev`.

Debería desplegarse una página web con el título "Cisco ATA 186 Configuration" en la parte superior, y una gran cantidad de opciones abajo. Las opciones que deben modificarse para obtener una configuración compatible con el * son:

- `UID0`: corresponde al nombre de usuario para la línea número 1. De acuerdo a la configuración del ejemplo, debería setearse en `101`.
- `PWD0`: corresponde a la contraseña para la línea número 1. Setear `gonchi`.
- `UID1`: igual que para `UID0`, pero para la línea número 2.
- `PWD1`: igual que para `PWD0`, pero para la línea número 2.
- `GkOrProxy`: configurara aquí la dirección IP del *.
- `UseSIP`: setear en `1` para indicar que se usará SIP y no H.323.
- `SIPRegOn`: setear en `1` para que el ATA envíe mensajes de registro al *.
- `AudioMode`: involucra muchas opciones para la configuración del audio en las líneas. En particular nos interesa modificar un solo bit, que deshabilita la supresión de silencios (al igual que en X-Lite). Debería setearse el siguiente valor `0x00140014`.

Notar que las demás opciones pueden dejarse en sus valores por defecto.

Luego de haber finalizado la configuración, presionando el botón de "Apply", el ATA debería reiniciarse salvando los cambios y ambas líneas deberían registrarse como usuarios SIP.

5.3. Pruebas NAT

Se llevaron a cabo pruebas en diversos escenarios NAT con el objetivo de verificar y entender la funcionalidad de * para llamadas SIP bajo estas restricciones. En esta sección se describen las diferentes topologías de prueba armadas y las configuraciones realizadas a nivel de la PBX, *routers* NAT y clientes SIP. Mediante el Ethereal, se capturaron los mensajes SIP y RTP de forma de verificar los cambios realizados a los paquetes IP, tanto por los *routers* NAT como por *.

5.3.1. Escenario 1

En este primer escenario de pruebas se tienen dos redes IP interconectadas a través de un *router* NAT modelo Linksys. La red pública se conecta al puerto WAN, mientras que la red privada se implementó mediante un *hub* conectado a uno de los puertos LAN del *router*. El esquema general de la topología se muestra en la figura 5.9.

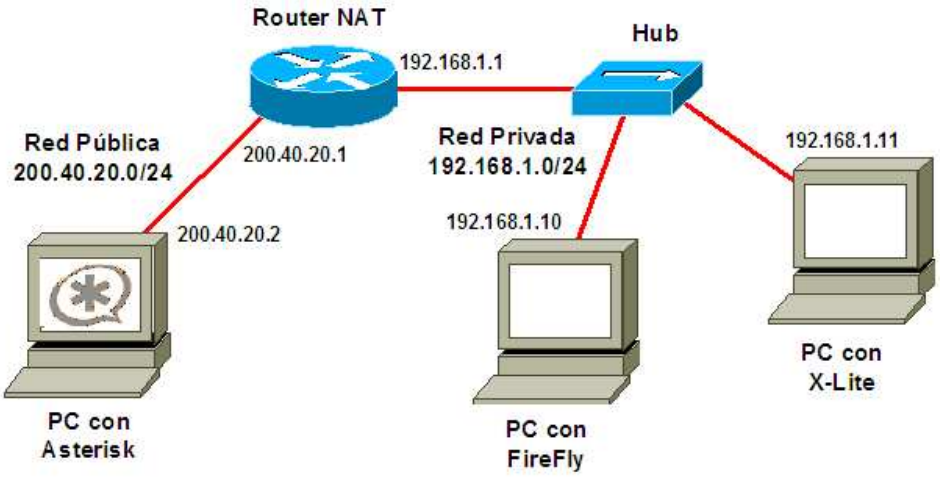


Figura 5.9: Primer escenario de pruebas.

Se observa que en este escenario, el * se encuentra en la red pública, mientras que los clientes SIP se encuentran en la red privada comunicándose vía NAT con el exterior. La dirección de la red pública es `200.40.20.0/24`, mientras que la red privada es `192.168.1.0/24`. La asignación de direcciones IP para ambas redes se muestra en la tabla 5.1.

Dispositivo	Dirección IP
Interfaz WAN del *router*	200.40.20.1
Interfaz LAN del *router*	192.168.1.1
Asterisk	200.40.20.2
PC con *softphone* Firefly	192.168.1.10
PC con cliente X-Lite	192.168.1.11

Tabla 5.1: Asignación de direcciones IP para el primer escenario de pruebas NAT.

En estas condiciones, para que los clientes SIP dentro la red privada puedan registrarse al * para realizar llamadas, se debe agregar la sentencia `nat=yes` en la configuración de cada *user* o *peer* en `sip.conf`. Para aquellos clientes con esta opción configurada, durante la transacción de registro, * ignora la información de contacto del cliente SIP contenida en los campos `Contact` o `Via` y toma la dirección IP y puerto directamente de los encabezados IP y UDP del mensaje REGISTER recibido.

A continuación se muestra el paquete IP correspondiente al REGISTER enviado por el *softphone* X-Lite. La captura fue realizada del lado de la red pública. Los aspectos interesantes son en primer lugar, la información IP de origen y destino. Se observa como dirección de origen a la IP de la interfaz del *router* contra la red pública, que es donde * deberá enviar los paquetes SIP cuando desee comunicarse con el X-Lite. Por otro lado, se observa que el puerto UDP de origen es 15060, por más que el *softphone* tenga configurado 5060 como puerto SIP. Esto se debe al funcionamiento del *router* NAT, que modifica el puerto, y podrá entregar correctamente los paquetes de la red pública al *softphone* en la red privada, a partir de la siguiente biyección:

`200.40.20.1:15060⇔192.168.1.1:5060`

Para el correcto funcionamiento del mecanismo NAT, la transacción debe ser iniciada por un cliente dentro de la red privada (porque iniciando desde la red pública, no se puede acceder a la red privada ya que el NAT nunca estableció la biyección), pero esto no es un impedimento ya que el registro en SIP es siempre iniciado por los clientes.

```
Internet Protocol, Src Addr: 200.40.20.1 (200.40.20.1),
                   Dst Addr: 200.40.20.2 (200.40.20.2)
User Datagram Protocol, Src Port: 15060 (15060), Dst Port: 5060 (5060)
Session Initiation Protocol
    Request-Line: REGISTER sip:200.40.20.2 SIP/2.0
    Message Header
        Via: SIP/2.0/UDP 192.168.1.11:5060;rport;branch=z9hG4bK28FA381F9
        From: fede <sip:100@200.40.20.2>;tag=3799245353
        To: fede <sip:100@200.40.20.2>
        Contact: "fede" <sip:100@192.168.1.11:5060>
        Call-ID: FFFB6602E6724B34808D5C11D30CED9D@200.40.20.2
        CSeq: 16615 REGISTER
        Expires: 1800
        Max-Forwards: 70
        User-Agent: X-Lite release 1103a
        Content-Length: 0
```

Observar también como no se modifica la información generada por el X-Lite en los campos `Via` y `Contact` : `192.168.1.11:5060`. Dado que * sabe que el cliente se encuentra detrás del NAT, descarta esta información de contacto y toma la información a nivel de IP/UDP. La salida en consola de `sip show peers` luego de los registros de los clientes detrás del NAT es:

```
asterisk*CLI> sip show peers
Name/username    Host            Dyn Nat ACL Mask             Port     Status
103/103          (Unspecified)    D          255.255.255.255  -1       Unmonitored
102/102          200.40.20.10     D   N      255.255.255.255  5060     Unmonitored
101/101          (Unspecified)    D          255.255.255.255  -1       Unmonitored
100/100          200.40.20.10     D   N      255.255.255.255  15060    Unmonitored
4 sip peers [4 online , 0 offline]
```

Por lo tanto, el mecanismo descrito es el que permite en este tipo de escenarios que la señalización SIP se intercambie correctamente entre las puntas. Otro gran tema es el intercambio del flujo RTP que utiliza otros puertos UDP asignados dinámicamente.

En las transacciones de inicio de llamadas, con los intercambios de mensajes INVITE y 200 OK entre los clientes y *, se procesa la información SDP de cada una de las partes. En particular, se informa la dirección IP y puerto UDP donde se intercambiará RTP. Estando * en la red pública, publica una dirección alcanzable por los clientes dentro del NAT. Sin embargo, la información recibida por * de los clientes incluye direcciones IP privadas ya que el *router* NAT no modifica para nada el SDP. En primera instancia, es de esperar que si lo dos clientes se encuentran en la red privada, ninguno de ellos escuche audio ya que el RTP de ambos llegará a *, pero éste no podrá entregarlo a destino. Si uno de los clientes estuviera en la red pública, sólo él podría escuchar audio.

¿Cómo resuelve * este inconveniente?. Consciente de la existencia de un NAT, aguarda a recibir audio del cliente en la red privada, dejando que abra el agujero en el NAT. A partir de los mensajes RTP recibidos, actualiza la dirección IP y puerto del origen (que serán la IP del *router* y algún puerto que el NAT haya asignado) y por lo tanto podrá entregar su RTP sin inconvenientes. Este mecanismo se encuentra implementado en el archivo fuente `rtp.c`, en la función `ast_rtp_read` que se encarga de procesar el flujo RTP entrante para las llamadas en ejecución.

```
if (rtp->nat) {
                /* Send to whoever sent to us */
                if ((rtp->them.sin_addr.s_addr != sin.sin_addr.s_addr) ||
                    (rtp->them.sin_port != sin.sin_port)) {
                        memcpy(&rtp->them, &sin, sizeof(rtp->them));
                        ast_log(LOG_DEBUG, "RTP NAT: Using address %s:%d\n",
                        ast_inet_ntoa(iabuf, sizeof(iabuf), rtp->them.sin_addr),
                        ntohs(rtp->them.sin_port));
                }
        }
```

Se observa que en `rtp->them` se almacena la información de IP y puerto del cliente mientras que en `sin` se tiene la dirección IP y puerto origen de quien envió un paquete RTP. En `rtp->them`, a partir de el procesamiento del SDP recibido, habrá una dirección privada. Por lo tanto se compara si hay diferencia a nivel de dirección IP o puerto y en dicho caso se actualiza la información para poder enviar el flujo de medios a destino.

Para la validación del funcionamiento en este escenario se llevaron a cabo llamadas con dos clientes dentro de la red privada y con un cliente a cada lado del NAT. El funcionamiento es correcto en todos los caso siempre y cuando se configure `canreinvite=no` en `sip.conf`, deshabilitando la posibilidad de intercambiar RTP entre las puntas.

En último lugar, es importante aclarar que para este escenario no fue necesario realizar ningún tipo de configuración especial en el router Linksys. Simplemente se lo configuró para que

opere como *gateway* y de esta forma brinde la funcionalidad NAT. Para la implementación de la WAN se le configuró su interfaz específica con una dirección estática.

5.3.2. Escenario 2

En esta nueva topología, se invierten los roles. Nuevamente un *router* NAT separa un red pública de una red privada, pero en este caso el * se encuentra detrás del NAT y los clientes afuera. El esquema de la red se muestra en la figura 5.10.

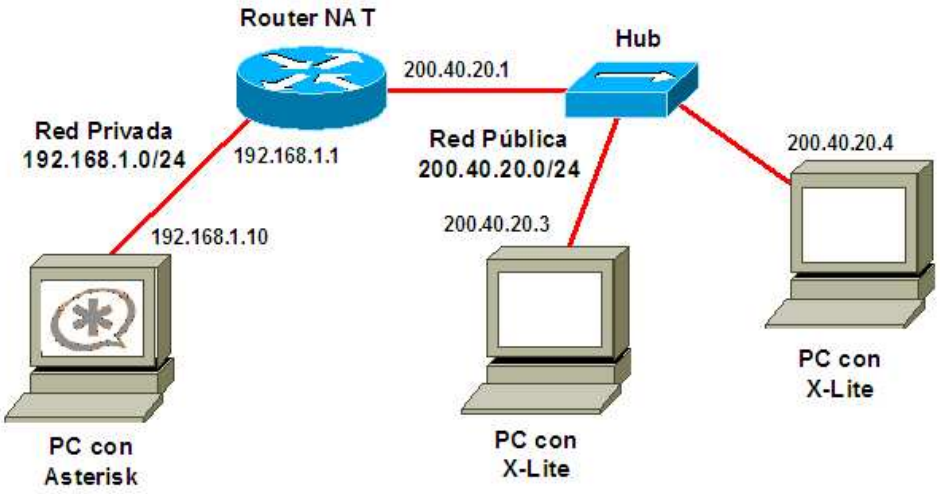


Figura 5.10: Segundo escenario de pruebas.

Al igual que en el escenario anterior, la dirección de la red pública es `200.40.20.0/24`, mientras que la red privada es `192.168.1.0/24`. La asignación de direcciones IP para ambas redes se muestra en la tabla 5.2.

Dispositivo	Dirección IP
Interfaz WAN del *router*	200.40.20.1
Interfaz LAN del *router*	192.168.1.1
Asterisk	192.168.1.10
PC con *softphone* X-Lite	200.40.20.3
PC con cliente X-Lite	200.40.20.4

Tabla 5.2: Asignación de direcciones IP para el segundo escenario de pruebas NAT.

En el caso anterior, estando * en la red pública, los clientes podían iniciar sin problemas las transacciones de registro. Para esta situación, no pueden enviar un REGISTER a la dirección IP privada del * por lo cual surge un nuevo problema a resolver. El correcto funcionamiento se logra mediante la configuración de un *port-forwarding* en el *router* Linksys. Se debe fijar que

todos los paquetes recibidos por el *router* en su interfaz WAN, destinados al puerto UDP 5060, se envíen a la dirección 192.168.1.10 correspondiente al *. Bajo el supuesto de que no haya más de un servidor en la red privada donde los clientes quieran registrarse, este método soluciona la problemática anterior.

Desde el punto de vista de la configuración de los clientes, se deberá definir 200.40.20.1 (dirección IP de la interfaz del *router* a la WAN), como dirección del *proxy* donde se registrarán. Una vez que los mensajes SIP llegan al *router*, mediante el `port-forwarding` al 5060 serán encaminados correctamente a la PBX. También es importante notar que no es necesario configurar `nat=yes` en este caso, ya que * podrá responderles desde la red privada con la dirección pública que reciba en los mensajes SIP.

Desde el punto de vista de los mensajes SIP generados por * estando detrás de un NAT, debe agregarse configuración adicional para que las transacciones SIP se completen exitosamente. Teniendo en cuenta la asignación de direcciones de la red de pruebas, en `sip.conf` deben configurarse las siguientes variables:

```
[general]

...
externip = 200.40.20.1
localnet = 192.168.1.0/255.255.255.0
...
```

El campo `externip` define la dirección IP de contacto que deberá fijar el * en sus mensajes SIP (encabezados `Contact` y `Via`). Por lo tanto, cuando * deba enviar mensajes SIP a clientes en la red pública, modificará los encabezados sustituyendo su dirección privada por la IP de la interfaz WAN del *router* NAT. Esta modificación a los encabezados es fundamental para que * pueda recibir respuestas de *peers* en la red pública que no se hayan registrado y probablemente no sepan que * se encuentra detrás de un NAT.

Es claro que estas modificaciones solo competen para transacciones con clientes fuera de la red privada. Para poder llevar a cabo esta distinción es que se configura el parámetro `localnet` que define la dirección de la red privada en la cual se encuentra el *. Antes de responder a una solicitud o directamente enviar una él mismo, * verificará si el destino se encuentra también dentro del NAT. En ese caso no deberán realizarse modificaciones a los encabezados `Contact` y `Via`.

Como ejemplo, en la figura 5.11 se muestra el flujo de mensajes para el establecimiento de una llamada entre los clientes fuera del NAT. Dicho diagrama fue generado a partir de una captura con Ethereal dentro de la red privada.
En el mensaje número 6, correspondiente al INVITE que * envía al terminal llamado para establecer el canal de salida, se observan las modificaciones mencionadas en los encabezados.

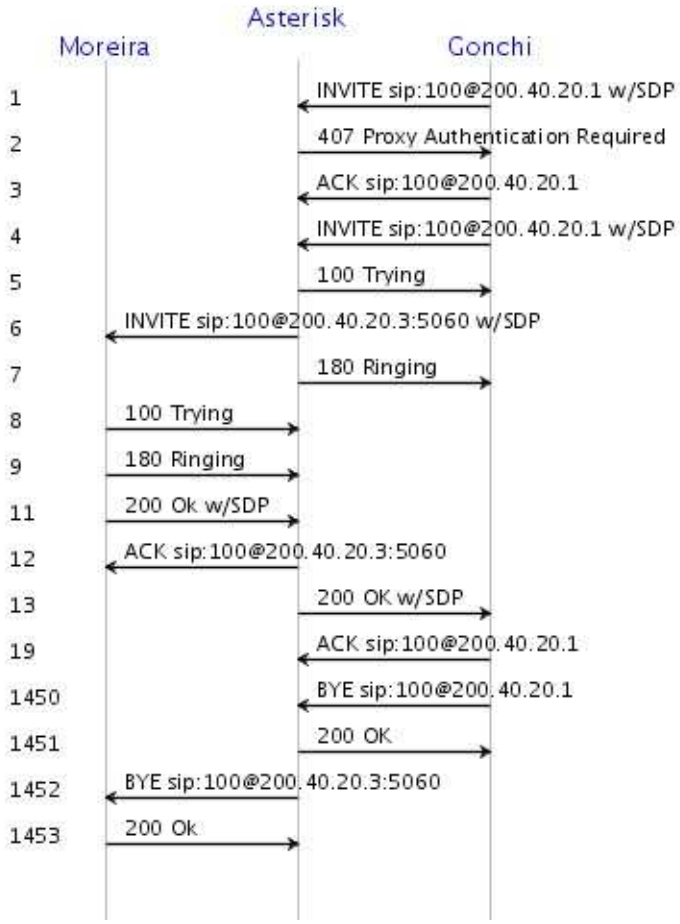


Figura 5.11: Una llamada en el segundo escenario.

Notar que también se modifica la dirección IP informada en el SDP.

```
Internet Protocol, Src Addr: 192.168.1.10 (192.168.1.10),
                   Dst Addr: 200.40.20.3 (200.40.20.3)
User Datagram Protocol, Src Port: 5060 (5060), Dst Port: 5060 (5060)
Session Initiation Protocol
    Request-Line: INVITE sip:100@200.40.20.3:5060 SIP/2.0
    Message Header
        Via: SIP/2.0/UDP 200.40.20.1:5060;branch=z9hG4bK72094683
        From: "Mauge" <sip:102@200.40.20.1>;tag=as38474cf9
        To: <sip:100@200.40.20.3:5060>
        Contact: <sip:102@200.40.20.1>
        Call-ID: 36aeed5628a1487e191660693d70934a@200.40.20.1
        CSeq: 102 INVITE
        User-Agent: Asterisk PBX
        Date: Mon, 18 Jul 2005 20:13:09 GMT
        Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER
```

```
        Content-Type: application/sdp
        Content-Length: 211
    Message body
        Session Description Protocol
            Session Description Protocol Version (v): 0
            Owner/Creator, Session Id (o): root 8042 8042 IN IP4 200.40.20.1
            Session Name (s): session
            Connection Information (c): IN IP4 200.40.20.1
            Time Description, active time (t): 0 0
            Media Description, name and address (m): audio 18838 RTP/AVP 3 101
            Media Attribute (a): rtpmap:3 GSM/8000
            Media Attribute (a): rtpmap:101 telephone-event/8000
            Media Attribute (a): fmtp:101 0-16
            Media Attribute (a): silenceSupp:off - - - -
```

Los aspectos hasta aquí descritos permiten resolver la señalización SIP, con clientes en la red pública y también en la red privada. En lo que respecta al flujo RTP, para que los clientes de la red pública puedan enviar audio hacia el *, se debe configurar un nuevo *port-forwarding* en el *router*. En este caso, el rango de puertos que * asigna dinámicamente para los flujos RTP debe direccionarse a la IP 192.168.1.10. * permite configurar este rango de puertos, con lo cual podrá restringirse de acuerdo a lo permitido por la aplicación particular. El archivo de configuración para ello es `rtp.conf` que se muestra a continuación con el rango del 10000 al 20000 habilitado:

```
; RTP Configuration
;
[general]
;
; RTP start and RTP end configure start and end addresses
;
rtpstart=10000
rtpend=20000
```

Al igual que para el primer escenario, las pruebas de funcionamiento con las configuraciones mencionadas anteriormente fueron satisfactorias, siempre y cuando el flujo RTP pase por *.

5.3.3. Escenario 3

El último escenario de estudio reúne elementos de las dos configuraciones anteriores. Se tiene el * detrás de un *router* NAT conectado a la red pública y clientes SIP detrás de otro NAT. A la red privada donde se encuentra el * y uno de los clientes se le denomina *Red Privada 1*, mientras que a la otra red privada se le denomina *Red Privada 2*. El esquema de dicha topología se muestra en la figura 5.12.
La red pública tiene la dirección 200.40.20.0/24, la red privada 1 192.168.1.0/24 y la red privada 2 192.168.0.0/24. La asignación completa de direcciones se presenta en la tabla 5.3:
Se colocó a través de un *hub* una máquina adicional para poder realizar capturas Ethereal en la red pública. Su dirección es 200.140.20.3.

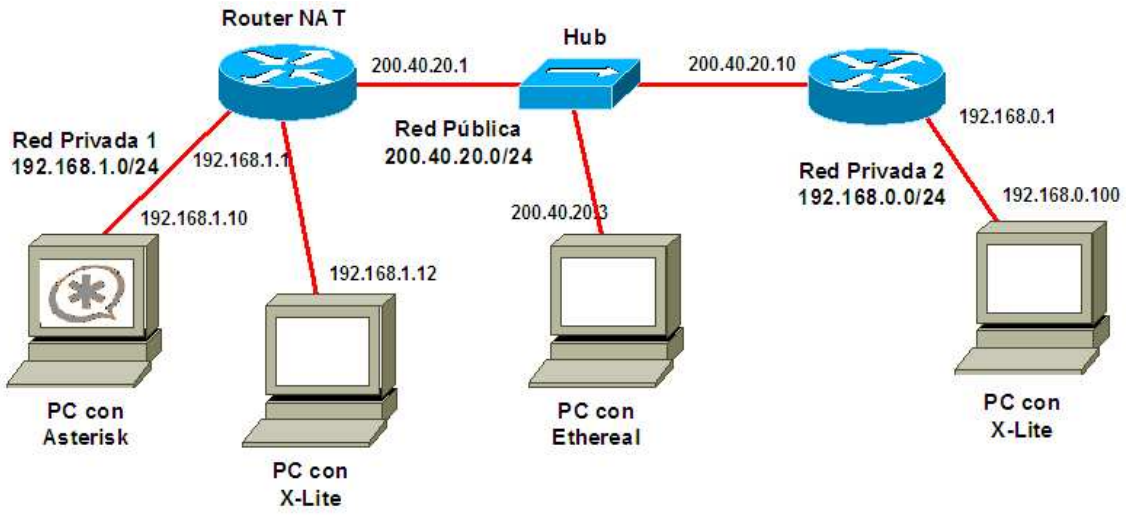


Figura 5.12: Tercer escenario de pruebas

Dispositivo	Dirección IP
Interfaz WAN del *router* 1	200.40.20.1
Interfaz LAN del *router* 1	192.168.1.1
Asterisk	192.168.1.10
PC con *softphone* X-Lite	192.168.1.12
Interfaz WAN del *router* 2	200.40.20.10
Interfaz LAN del *router* 2	192.168.0.1
PC con cliente X-Lite	192.168.0.100

Tabla 5.3: Asignación de direcciones IP para el tercer escenario de pruebas NAT.

A continuación se describe la configuración relevante para cada uno de los equipos, que incluye lo visto en los casos anteriores. Para el *, que opera detrás de un NAT se debe configurar en la sección `[general]` del `sip.conf` las variables `externip=200.40.20.1` y `localnet=192.168.1.1/255.255.255.255`. Para el *softphone* que se encuentra en la red privada 1, se configura directamente la dirección privada del * ya que no tendrá que cursar la red pública para alcanzarlo. En lo que respecta al *router* 1, se lo configura como *gateway* con interfaz WAN estática 200.40.20.1. Se deben habilitar los *port-forwarding* de 5060 UDP y el rango definido para RTP hacia el * (192.168.1.10). El *router* 2 solo debe ser configurado como *gateway* con interfaz WAN estática 200.40.20.10. Finalmente, en lo que respecta al *softphone* dentro de la red privada 2, se le debe configurar la dirección 200.40.20.1 para alcanzar al *. Además en `sip.conf` para el usuario SIP correspondiente a dicho terminal, se debe agregar la variable `nat=yes`.

En la figura 5.13 se observa el flujo de mensajes SIP capturados en la red pública, para una

llamada iniciada por el cliente de la red privada 2, hacia el otro cliente en la red privada 1. Notar como la transacción de inicio de sesión entre * y el softphone llamado no se observa aquí, ya que se resuelve completamente dentro de la red privada 1.

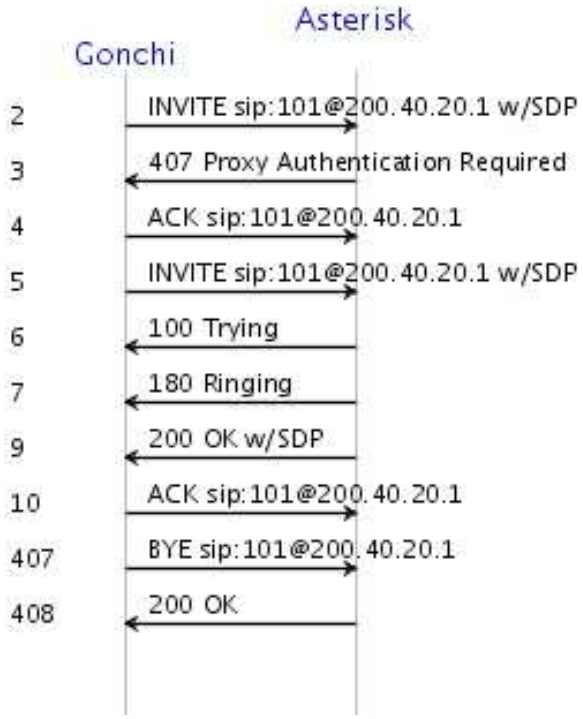


Figura 5.13: Una llamada en el tercer escenario, capturada desde la red pública.

Más interesante aún es observar la transacción completa para la misma llamada, pero capturada desde la red privada 1.
Si consideramos un fragmento del *payload* SDP enviado por * al cliente de la red privada 2, digamos para el 200 OK (mensaje número 25), observamos que modifica la dirección que informa para el flujo RTP:

```
Session Description Protocol Version (v): 0
Owner/Creator, Session Id (o): root 8446 8446 IN IP4 200.40.20.1
Session Name (s): session
Connection Information (c): IN IP4 200.40.20.1
Time Description, active time (t): 0 0
```

Sin embargo, para un SDP similar enviado al cliente en su misma red, como puede ser en el INVITE (mensaje número 17), la dirección de contacto no se modifica ya que * es capaz de reconocer que se encuentra detrás del mismo NAT que el.

```
Session Description Protocol Version (v): 0
Owner/Creator, Session Id (o): root 8446 8446 IN IP4 192.168.1.10
```

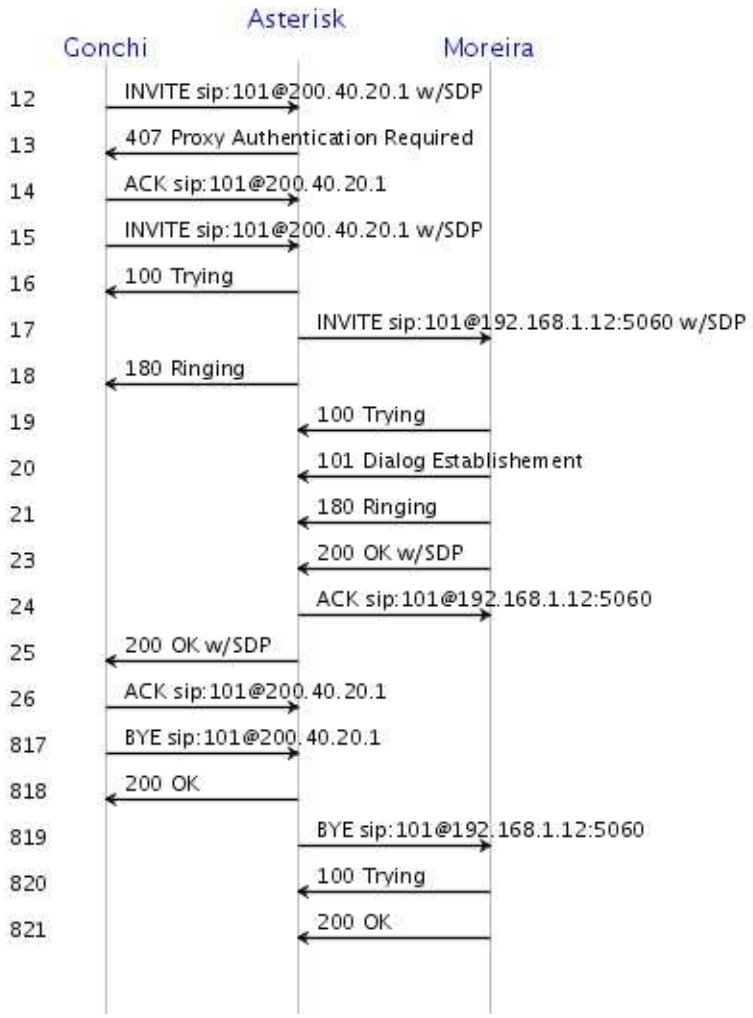


Figura 5.14: Una llamada en el tercer escenario, capturada desde la red privada 1.

```
Session Name (s): session
Connection Information (c): IN IP4 192.168.1.10
Time Description, active time (t): 0 0
```

Era de esperar un correcto funcionamiento en este escenario a partir de las pruebas satisfactorias realizadas en los primeros dos. Para validar los resultados en una topología similar, pero tal vez un poco más realista desde el punto de vista de las aplicaciones que se puedan presentar, sustituimos la red pública implementada como un enlace físico por Internet. El esquema de la red se muestra en la figura 5.15.
En los tres casos, las salidas a Internet de las redes locales era a través de servicios ADSL. Las pruebas nuevamente fueron satisfactorias, pero en este contexto la calidad de las llamadas se veía afectada por el retardo y el excesivo ancho de banda requerido por algunos codecs.

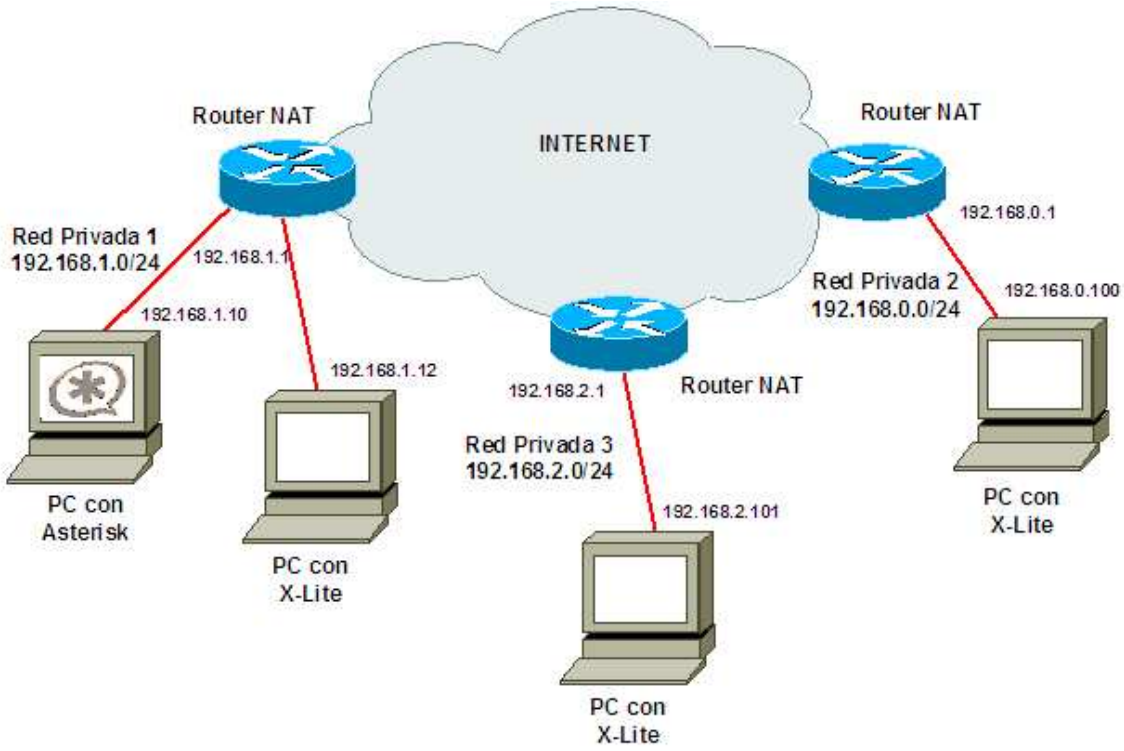


Figura 5.15: Escenario similar, con Internet como red pública.

Para llamadas utilizando el codec `gsm` la calidad era buena, mientras que con `PCM uLaw` el ancho de banda no era suficiente y se escuchaba demasiado entrecortado haciendo imposible la comunicación.

5.4. Manejo avanzado de llamadas: las transferencias

En esta sección se describen los aspectos vinculados a la implementación de transferencias en llamadas SIP. Por un lado, se presentan las recomendaciones de la IETF relativas a los mensajes entre los UAs para llevar a cabo transferencias directas (incondicionales) y asistidas. Se estudia como * y su arquitectura de manejo de canales de entrada y salida, se adecúan a dichas recomendaciones para manejar las solicitudes de transferencias de los clientes.

Por otro lado, dicha arquitectura de dos canales por llamada permiten que * pueda implementar las transferencias sin necesidad de señalización externa, solamente *puenteando* canales. Esta implementación tiene la ventaja de ser independiente de la tecnología del canal y traslada la complejidad desde el cliente al servidor.

En los distintos escenarios de transferencias que se describen a continuación, intervienen tres actores con distintos roles:

- **Usuario transferido**: quién será transferido al *Usuario destino de la transferencia*.
- **Usuario que transfiere**: quién inicia al transferencia.
- **Usuario destino de la transferencia**: quién no era parte de la llamada, pero tras la transferencia queda comunicado con el *Usuario transferido*.

5.4.1. Transferencia directa

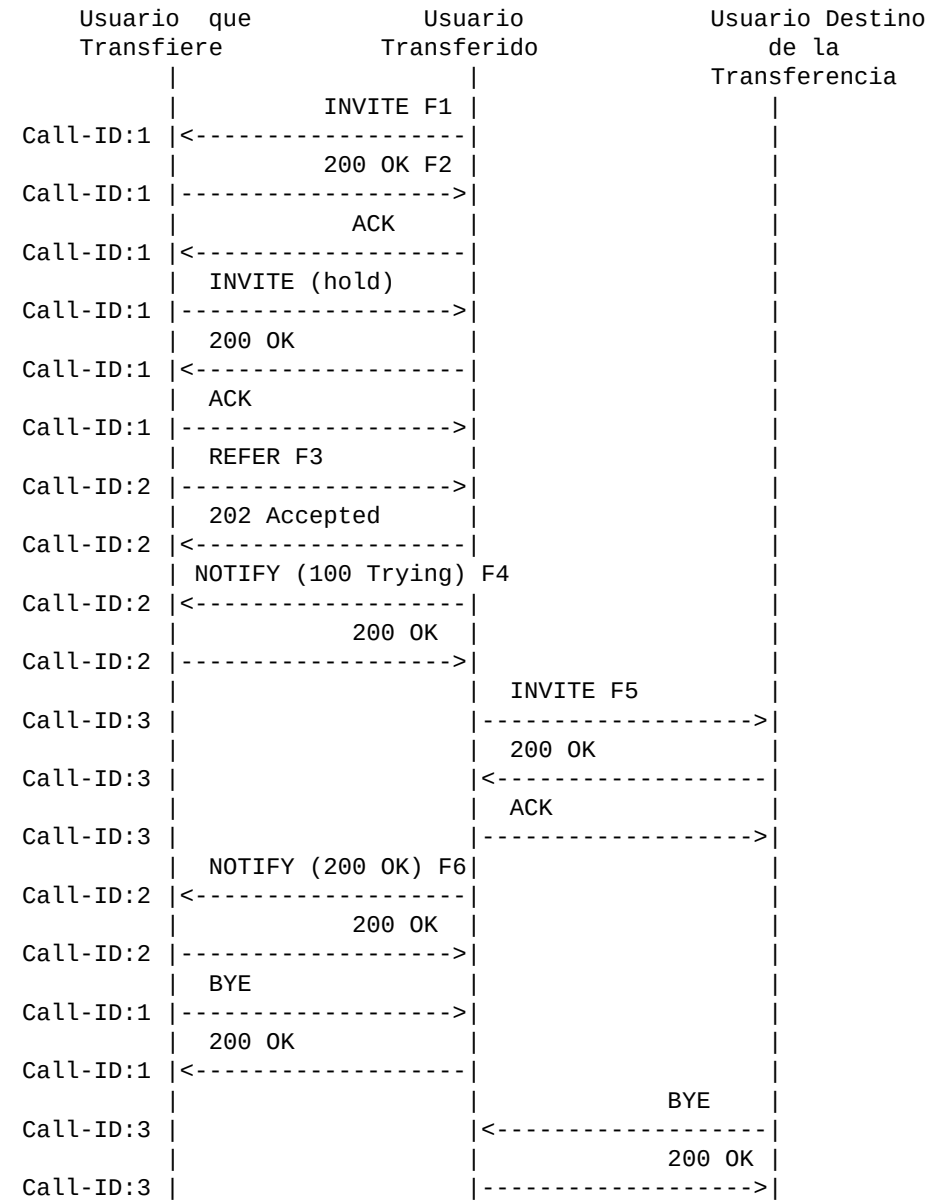
En primer lugar analizaremos la transferencia más sencilla, conocida como transferencia directa o incondicional. En dicho escenario el *Usuario transferido* y el *Usuario que transfiere* se encuentran comunicados en una llamada. En cierto instante el *Usuario que transfiere* decide transferir al *Usuario transferido*, pasándole la llamada al *Usuario destino de la transferencia*. En el momento que se inicia la trasferencia, el *Usuario transferido* queda en espera escuchando música de espera. En ese lapso el *Usuario que transfiere* señaliza el número del destino a la PBX. Si la transferencia se resuelve con éxito, el *Usuario que transfiere* deja la llamada, en caso de error retorna a la llamada con el *Usuario transferido*.

En uno de los *Internet Drafts* del *Sipping Workgroup* [11], se especifica en detalle la secuencia de mensajes SIP necesarios para implementar dicha funcionalidad. Se recomienda la utilización de mensajes REFER, para que el *Usuario transferido* pueda enviar un INVITE al *Usuario destino de la transferencia*.

En resumen, para implementar una transferencia directa se recomienda seguir los siguientes pasos:

1. El *Usuario que transfiere* pone al *Usuario transferido* en espera. Para ello se envía un nuevo INVITE con el mismo `Call-ID` que ya identificada su llamada actual y con `sendonly` como atributo especial en el SDP.
2. El *Usuario que transfiere* le envía un REFER con la información de contacto del *Usuario destino de la transferencia*.
3. El *Usuario transferido* inicia una nueva sesión con el *Usuario destino de la transferencia* y tras la finalización de esta transacción de inicio de llamada, informa al *Usuario que transfiere* si ha sido exitosa o no, mediante un mensaje NOTIFY.
4. En el caso de que la transferencia haya sido satisfactoria, el *Usuario transferido* termina la sesión con el *Usuario que transfiere*.

A continuación se muestra el flujo de mensajes SIP para implementar la secuencia descrita arriba:



Como mensajes interesantes a analizar, tenemos en primer lugar el REFER. Observar el encabezado `Refer-To` donde se informa la URI del *Usuario destino de la transferencia*.

```
REFER sip:transferido@undominio.com SIP/2.0
Via: SIP/2.0/TLS proxy.otrodominio.com;branch=z9hG4bKna9
Max-Forwards: 70
To: <sip:transferido@undominio.com;grid=3413kj2ha>
From: <sip:quetransfiere@otrodominio.com>;tag=1928301774
Call-ID: a84b4c76e66710
CSeq: 314159 REFER
```

```
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, NOTIFY
Refer-To: <sip:destino@ultimodominio.com>
Contact: <sip:quetransfiere@otrodominio.com;grid=723jd2d>
Content-Length: 0
```

En segundo lugar, se tiene el primer NOTIFY (identificado como mensaje F4) que informa al *Usuario que transfiere* que la transferencia está en proceso. Esto se implementa mediante el encabezado `subscription-state` con el valor `active`.

```
NOTIFY sip:quetransfiere@otrodominio.com SIP/2.0
Via: SIP/2.0/TLS proxy2.undominio.com;branch=z9hG4bKnas432
Max-Forwards: 70
To: <sip:quetransfiere@otrodominio.com>;tag=1928301774
From: <sip:transferidp@undominio.com;grid=3413kj2ha>
Call-ID: a84b4c76e66710
CSeq: 73 NOTIFY
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, NOTIFY
Event: refer
Subscription-State: active;expires=60
Content-Type: message/sipfrag
Content-Length: ...
```

Finalmente, luego de establecida la nueva sesión entre el *Usuario transferido* y el *Usuario destino de la transferencia*, se informa que fue exitosa al *Usuario que transfiere* a través de un nuevo NOTIFY (mensaje F6). En este caso el encabezado `subscription-state` tiene el valor `terminated`.

```
NOTIFY sip:quetransfiere@otrodominio.com SIP/2.0
Via: SIP/2.0/TLS proxy2.undominio.com;branch=z9hG4bKnas432
Max-Forwards: 70
To: <sip:quetransfiere@otrodominio.com>;tag=1928301774
From: <sip:transferidp@undominio.com;grid=3413kj2ha>
Call-ID: a84b4c76e66710
CSeq: 74 NOTIFY
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, NOTIFY
Event: refer
Subscription-State: terminated;reason=noresource
Content-Type: message/sipfrag
Content-Length: ...
```

De manera de verificar el comportamiento de * frente a transferencias directas implementadas con mensajes SIP (REFER-NOTIFY), se hicieron pruebas con el cliente Eyebeam que tiene una función propia para iniciar este tipo de transferencias. Para realizar la transferencia, simplemente hay que presionar el botón de TRANSFER y marcar la extensión de destino. Presionando TRANSFER nuevamente, se confirma el valor ingresado e inicia la transacción. En nuestro escenario de prueba, el *Usuario transferido* es un *softphone* X-Lite (IP 192.168.0.101) que se encuentra en una llamada con el Eyebeam (IP 192.168.0.100), *Usuario que transfiere*. El

Usuario destino de la transferencia es un *softphone* Linphone (IP 192.168.0.15) que se encuentra registrado al * (IP 192.168.0.10) con número de extensión 101.

Para analizar la transacción, se realizó una captura Ethereal filtrando por UDP. El flujo completo de mensajes para la transferencia se muestra en la figura 5.16.

En primer lugar se observa que el Eyebeam al iniciar el proceso de transferencia, pone en espera al X-Lite mediante un nuevo INVITE (indicado como mensaje número 170 en el diagrama) dirigido al *, con atributo `sendonly` en el SDP. Por lo tanto, * debe procesar este nuevo INVITE de manera acorde, reproduciendo música de espera en el canal existente entre * y el X-Lite.

Efectivamente, luego de analizar el código de la función `process_sdp` en el archivo fuente `chan_sip.c`, se tiene la siguiente condición que implementa la función descrita en el párrafo previo:

```
/* Turn on/off music on hold if we are holding/unholding */
            if (!net_addr_ishostnull(sin) && !sendonly) {
                ast_moh_stop(p->owner->bridge);
            } else {
                ast_moh_start(p->owner->bridge, NULL);
                if (sendonly)
                    ast_rtp_stop(p->rtp);
            }
```

Las funciones `ast_moh_start` y `ast_moh_stop` son las que inician y detienen la reproducción de música de espera respectivamente. La variable `sendonly` es asignada cuando se recorren los atributos SDP.

Luego comienza la transacción de transferencia cuando el Eyebeam envía el REFER (mensaje número 768) al *, con la información de contacto del Linphone en su encabezado `Refer-To`.

```
Request-Line: REFER sip:102@192.168.0.10 SIP/2.0
To: "Mauge"<sip:102@192.168.0.10>;tag=as3b1167c2
From: <sip:100@192.168.0.100:9187>;tag=7d67bd5c
Via: SIP/2.0/UDP 192.168.0.100:9187;branch=z9hG4bK
Call-ID: 096028990658a3f73f71dd0e3dd08440@192.168.0.10
CSeq: 3 REFER
Contact: <sip:100@192.168.0.100:9187>
Max-Forwards: 70
User-Agent: eyeBeam release 3002s stamp 15131
Refer-To: <sip:101@192.168.0.10>
Referred-By: fede<sip:100@192.168.0.10>
Content-Length: 0
```

A partir de este momento, donde * debe procesar el REFER para completar la transferencia es que aparece una primera diferencia con lo recomendado con la IETF. * envía un único mensaje

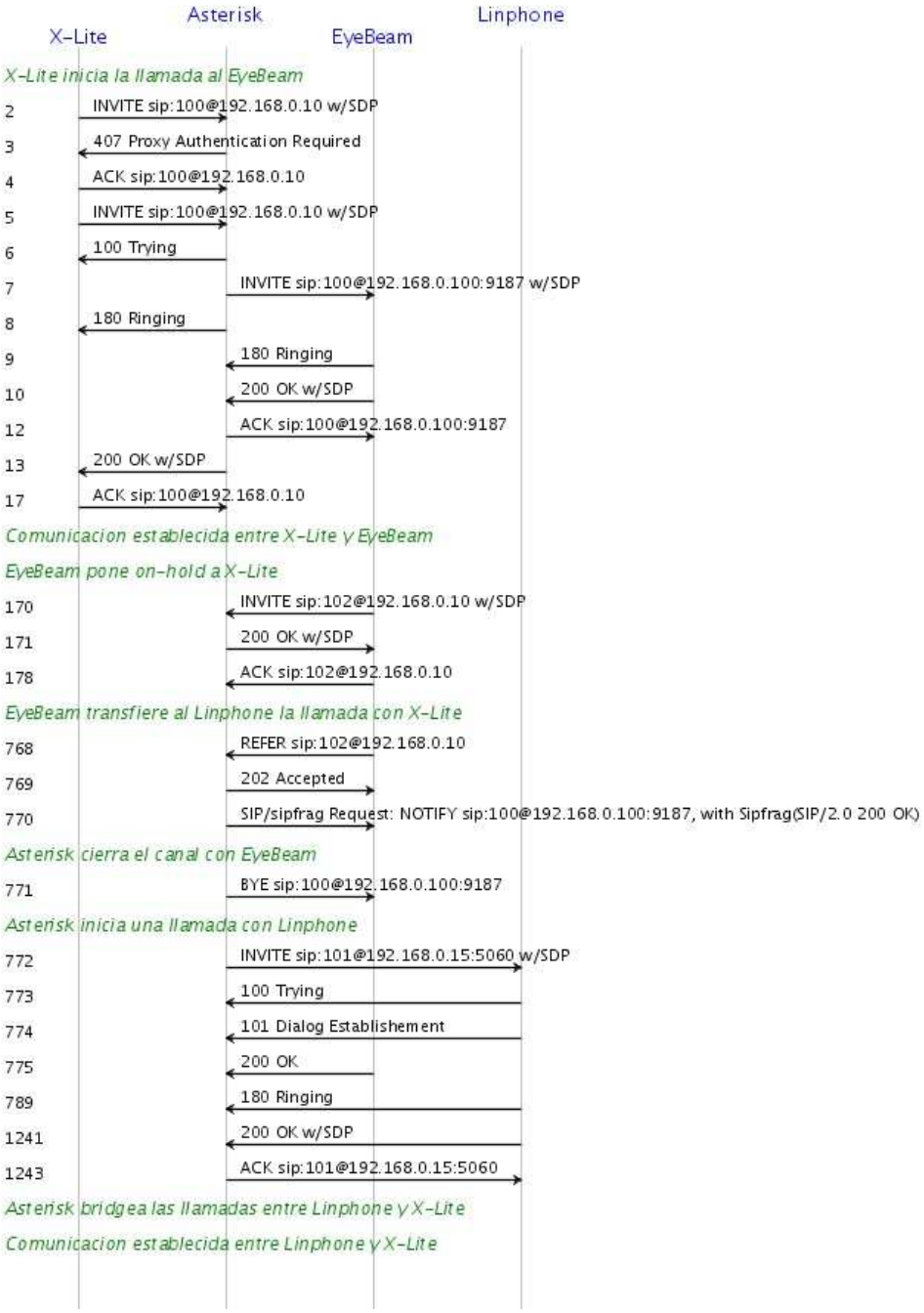


Figura 5.16: Transferencia directa implementada por el *softphone*.

NOTIFY al Eyebeam con el encabezado `subscription-state` y el valor `terminated`. Esto es antes de que se complete la transferencia y no se pasa por el estado intermedio de `active`.

```
Request-Line: NOTIFY sip:100@192.168.0.100:9187 SIP/2.0
Via: SIP/2.0/UDP 192.168.0.10:5060;branch=z9hG4bK3f173c34;rport
From: "Mauge"<sip:102@192.168.0.10>;tag=as3b1167c2
To: <sip:100@192.168.0.100:9187>;tag=7d67bd5c
Contact: <sip:102@192.168.0.10>
Call-ID: 096028990658a3f73f71dd0e3dd08440@192.168.0.10
CSeq: 103 NOTIFY
User-Agent: Asterisk PBX
Event: refer;id=3
Subscription-state: terminated;reason=noresource
Content-Type: message/sipfrag;version=2.0
Content-Length: 14
```

Esto presenta el problema de que se termina prematuramente la sesión entre el * y el Eyebeam mediante el BYE (mensaje número 771). Recién luego de este evento, en el mensaje número 772 es que se envía el INVITE al Linphone para completar la transferencia. En el ejemplo de la figura, la transferencia se completa de manera exitosa. Pero ante la eventualidad de que no pueda completarse (por ejemplo si el Linphone está ocupado), al haberse cortado la llamada con el Eyebeam también se terminará abruptamente la llamada con el X-Lite, mientras se encontraba escuchando la música en espera. Esto va en contra de las recomendaciones de la IETF, que sugieren mantener la sesión entre el *Usuario que transfiere* y el *Usuario transferido* hasta que la transferencia se lleve a cabo correctamente. En estas condiciones, en caso de tener problemas para transferir se puede retomar la sesión inicial.

Por otro lado, * brinda funcionalidad propia para realizar transferencias mediante el *puenteo* de canales y sin necesidad de señalización externa adicional intercambiada entre los terminales. Dado que basa en el manejo de canales como estructura abstracta, brinda la funcionalidad para todas las tecnologías que puedan implementarse en *.

A nivel de código, estas funciones se implementan en el archivo fuente `res_features.c`. Este archivo ha dejado obsoleto al antiguo `res_parking.c` que implementaba exclusivamente la funcionalidad de parqueo de llamadas. Ahora se le han agregado las transferencias directas y asistidas, entre otras funciones adicionales de PBX.

El archivo de configuración relevante para estas funciones es `features.conf`, donde en lo que respecta a las transferencias directas se puede definir la secuencia de dígitos a marcar para iniciarlas. A continuación se muestra un fragmento de dicho archivo, donde se ha configurado la secuencia `#1` para las transferencias directas.

```
[featuremap]
blindxfer => #1                         ; Blind transfer
disconnect => *0                 ; Disconnect
automon => *1                         ; One Touch Record
atxfer => *2                         ; Attended transfer
```

Otro aspecto de configuración que debe ser tenido en cuenta es a nivel del *dialplan*. Para aquellas extensiones que se desee permitirles la funcionalidad de transferir o ser transferidos, se deben configurar algunos argumentos adicionales en la aplicación Dial. A saber:

- `t`: permite que el usuario llamado pueda transferir la llamada, es decir, actuar como *Usuario que transfiere*
- `T`: permite que el usuario que llama pueda transferir la llamada, siendo ahora el llamado el *Usuario transferido*

La configuración de alguno de estos argumentos opcionales también restringe la posibilidad de encaminar el flujo RTP entre las puntas sin pasar por *. Por más que para los usuarios participando en la llamada se haya configurado `canreinvite=yes`, la posibilidad de transferir con funcionalidad de * fuerza a que el audio pase por la PBX.

Desde el punto de vista del funcionamiento y lo que perciben los usuarios es muy similar al caso anterior resuelto mediante mensajes SIP. En esta oportunidad el *Usuario que transfiere* marca la secuencia `#1` y se inicia la reproducción de música de espera para el *Usuario transferido*. Al mismo tiempo el *Usuario que transfiere* escucha el mensaje "Transfer", que le indica que su secuencia ha sido bien interpretada por la PBX y que debe marcar la extensión a donde transferir. Luego de que el * capture los DTMF correspondientes a la extensión podrá realizar la llamada al destino y *puentearla* con el canal del *Usuario transferido*.

En la figura 5.17 se muestra un flujo de mensajes SIP para una transferencia directa similar a la ya estudiada pero en este caso implementada por *. Un *softphone* Firefly (IP 192.168.0.100) es el *Usuario transferido*, un X-Lite (IP 102.168.0.101) es el *Usuario que transfiere* y finalmente el Linphone (192.168.0.15) es *Usuario destino de la transferencia*. El flujo se simplifica bastante respecto del caso anterior, ya que mucha funcionalidad antes implementada en la red, ahora la procesa * internamente. En primer lugar, no hay necesidad de un REINVITE para poner al Firefly en espera, al detectar * que se inicia una nueva transferencia automáticamente inicia la reproducción de música de espera en dicho canal. Más aún no hay necesidad de mensajes REFER y NOTIFY ya que * captura los tonos ingresados por el *Usuario que transfiere* y puede regenerar la cadena correspondiente a la extensión a donde transferir. Tampoco hay porque notificar al X-Lite del estado de la transferencia, ya que * centraliza todo el proceso y es quien conoce y mantiene la información del estado de los tres participantes.

5.4.2. Transferencia asistida

La transferencia asistida o consultada es una variación de la trasferencia directa. En este caso se incluye una sesión entre el *Usuario que transfiere* y el *Usuario destino de la transferencia* antes de que la transferencia realmente se lleve a cabo.

De acuerdo con las recomendaciones de implementación de la IETF, este tipo de transferencia puede lograrse con los elementos vistos para la transferencia directa. La secuencia recomendada es:

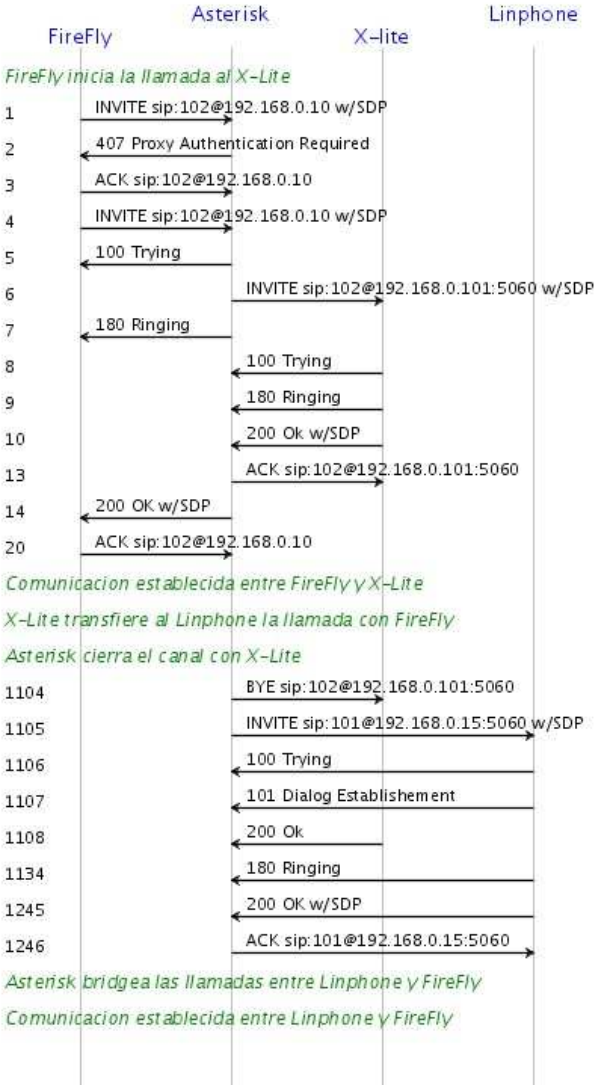


Figura 5.17: Transferencia directa implementada por *.

1. El *Usuario que transfiere* pone al *Usuario transferido* en espera. Para ello se envía un nuevo INVITE con el mismo `Call-ID` que ya identificada su llamada actual y con `sendonly` como atributo especial en el SDP.

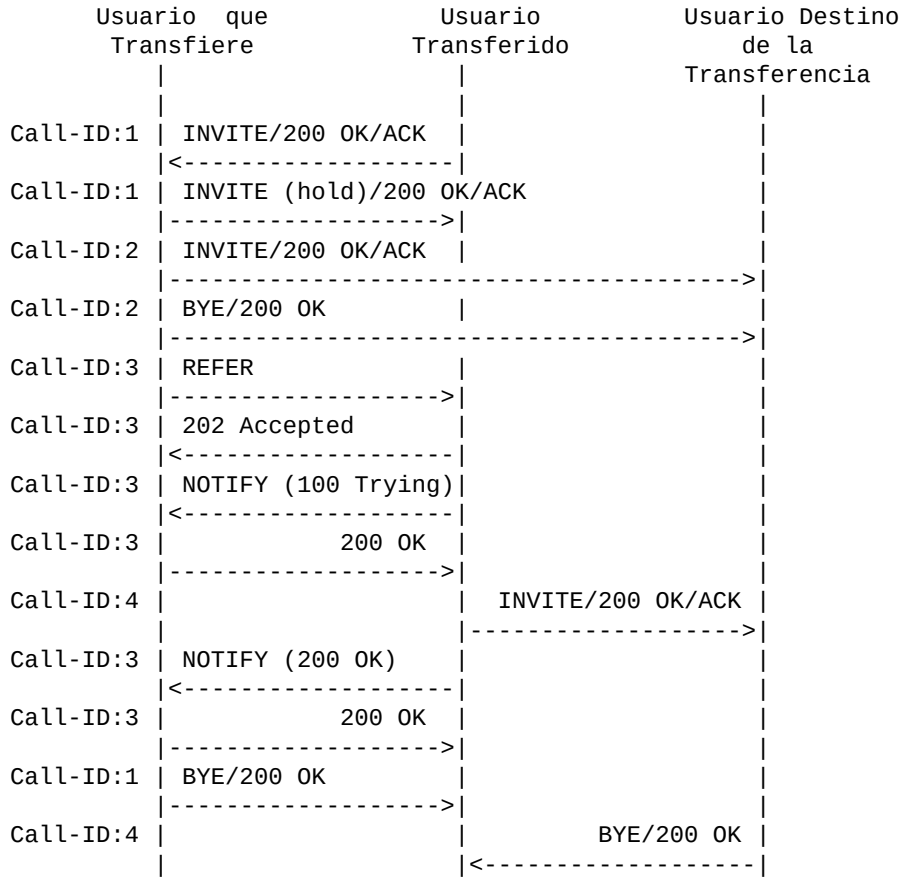
2. El *Usuario que transfiere* inicia una nueva sesión de consulta con el *Usuario destino de la transferencia*. Tras finalizar la comunicación se debe terminar la sesión.

3. El *Usuario que transfiere* le envía un REFER con la información de contacto del *Usuario destino de la transferencia*.

4. El *Usuario transferido* inicia una nueva sesión con el *Usuario destino de la transferencia* y tras la finalización de esta transacción de inicio de llamada, informa al *Usuario que transfiere* si ha sido exitosa o no, mediante un mensaje NOTIFY.

5. En el caso de que la transferencia haya sido satisfactoria, el *Usuario transferido* termina la sesión con el *Usuario que transfiere*.

El flujo de mensajes SIP que implementa dicha secuencia se muestra a continuación. Las transacciones no específicas de la transferencia han sido colapsadas en una sola línea por claridad.



En cuanto al soporte de * para esta funcionalidad, la transferencia asistida también se incluye en el paquete introducido en `res_features.c`. Para la configuración deben hacerse las mismas consideraciones respecto a los argumentos `t` y `T` de la aplicación `Dial`. En `features.conf` también se define la secuencia de dígitos para iniciar una transferencia asistida (ver el fragmento de configuración mostrado en la sección previa, variable `atxfer`).

La secuencia de mensajes SIP obtenida a partir de una captura Ethereal durante una transferencia asistida se muestra en la figura 5.18. Al igual que en el ejemplo anterior, el *softphone* Firefly (IP 192.168.0.100) es el *Usuario transferido*, un X-Lite (IP 102.168.0.101) es el *Usuario que*

transfiere y finalmente el Linphone (192.168.0.15) es *Usuario destino de la transferencia*. Nuevamente no hay necesidad de un REINVITE para poner en espera al Firefly, ni tampoco de los mensajes REFER y NOTIFY para el control de la transferencia.

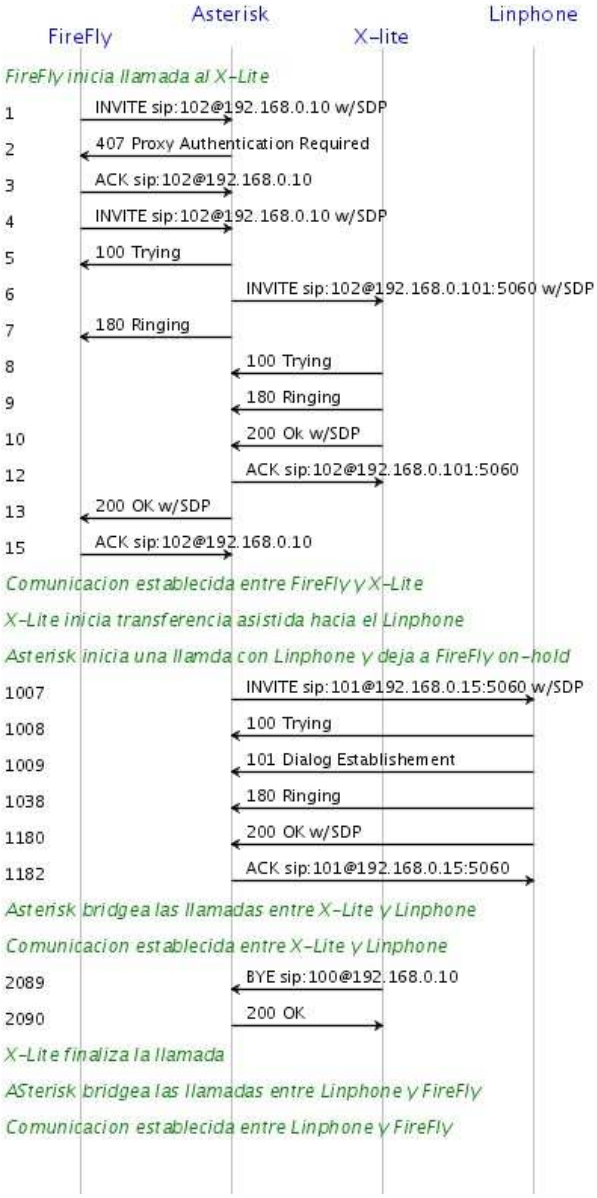


Figura 5.18: Transferencia asistida implementada por *.

En esta implementación a diferencia de lo recomendado por la IETF, la sesión de consulta no se termina completamente para luego iniciar la transferencia. Cuando el X-Lite corta su

llamada de consulta con el Linphone, * no cierra el canal creado con éste último. Directamente lo *puentea* con el canal que tenía establecido con el Firefly completando la transferencia. Esto es ventajoso ya que el Linphone no tiene que cortar para luego recibir una nueva llamada del Firefly, según está especificado en los documentos de la IETF. En este caso simplemente sigue en línea y * le conecta instantáneamente la llamada con el Firefly, lo que brinda un efecto en los usuarios mucho más similar al que daría una PBX tradicional.

6. IPV6 PARA EL CANAL SIP DE *

El protocolo IPv6, definido en la RFC 2460 [8], también conocido como IPng de *Next Generation Internet Protocol*, es una nueva versión de IP diseñada como evolución del actualmente difundido IPv4[1].

* no soporta el funcionamiento en redes IPv6. Luego, en la etapa de decisión acerca de qué funcionalidad extra podría brindarse a la PBX, resultó atractiva la idea de implementar el soporte para dicho protocolo (manteniendo, por supuesto, la compatibilidad hacia atrás con IPv4). Dicha extensión se acotó al canal SIP, ya que es éste el protocolo VoIP en el cual se basa el proyecto.

A efectos de cumplir este objetivo, se implementaron las modificaciones al canal SIP que permiten realizar llamadas en escenarios IPv6 puros, IPv4 puros, e IPv4/IPv6 *dual stack*.

En el presente capítulo se describen dichas modificaciones, luego de presentarse una breve introducción sobre el protocolo IPv6, mostrando sus ventajas sobre IPv4, sus modificaciones a nivel de encabezado y finalmente la nueva arquitectura de direcciones.

6.1. El protocolo IPv6

Como ya se mencionó, IPv6 es una nueva versión de IP diseñada como evolución de IPv4. Más aún, en IPv6 se han definido mecanismos de transición para que los usuarios puedan migrar sus sistemas de IPv4 a IPv6 de forma natural y sin problemas de interoperabilidad. El núcleo de protocolos de IPv6 se convirtió en estándar de la IETF en Agosto de 1998.

IPv6 fue diseñado para dar un paso evolutivo, no para establecer un cambio radical respecto a IPv4. Los elementos de IPv4 que funcionaban correctamente se han mantenido, el resto se ha eliminado. De acuerdo a la RFC 2460, los principales cambios en IPv6 respecto a IPv4 recaen en las siguientes categorías:

- **Expansión de las capacidades de direccionamiento**. IPv6 incrementa el tamaño de las direcciones IP de 32 a 128 bits para soportar mayor cantidad de niveles jerárquicos de direccionamiento, un número mucho mayor de nodos direccionables y un mecanismo sencillo de configuración automática de direcciones. La escalabilidad del enrutamiento *multicast* se ha mejorado mediante el agregado de un campo de *scope* (alcance) en las

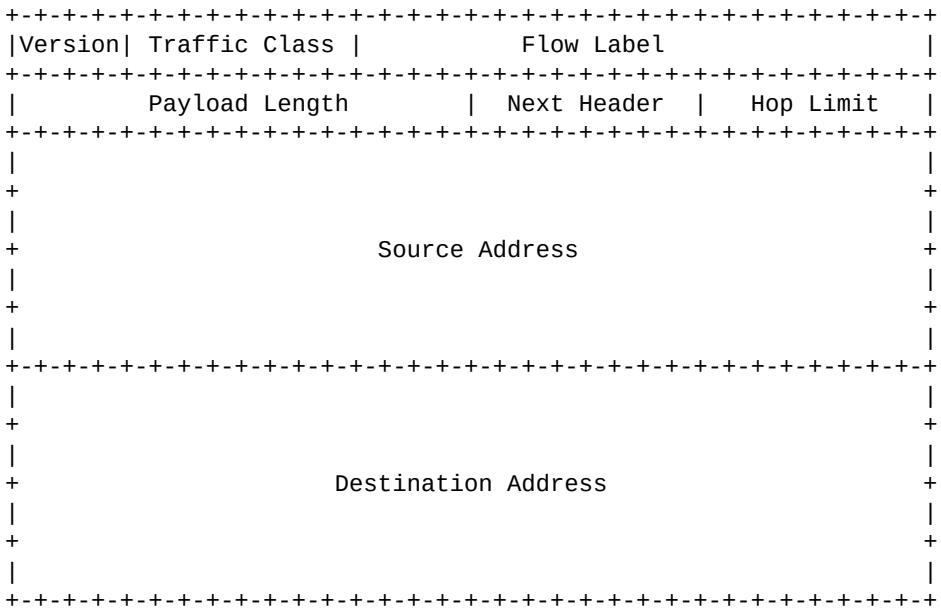
[1] Definido en la RFC 791 de Setiembre de 1981.

direcciones multicast. Finalmente, un nuevo tipo de dirección denominada *anycast* permite direccionar al nodo más cercano (en el sentido del protocolo de ruteo) dentro de un grupo predefinido.

- **Simplificación del encabezado**. Algunos campos del encabezado IPv4 se han eliminado o vuelto opcionales principalmente por dos razones: reducir el costo generalmente admitido del procesamiento de paquetes, y limitar el consumo de ancho de banda del encabezado IPv6.
- **Mejora en el soporte para extensiones y opciones**. Mejoras en la codificación de las opciones en el encabezado IPv6 hacen mucho más eficiente el *forwarding*, hacen menos exigentes las limitaciones sobre el largo de las opciones y permiten flexibilidad para el agregado de nuevas opciones a futuro.
- **Capacidad para el etiquetado de flujos**. Una nueva funcionalidad ha sido agregada para permitir el etiquetado de diferentes flujos de tráfico para los que el origen ha solicitado un tratamiento especial. Esto favorece las políticas de QoS y aplicaciones de tiempo real.
- **Capacidades de autenticación y seguridad**. Se han definido extensiones para soportar autenticación, verificación de integridad y opcionalmente la confidencialidad de los datos.

6.1.1. Encabezado IPv6 y extensiones

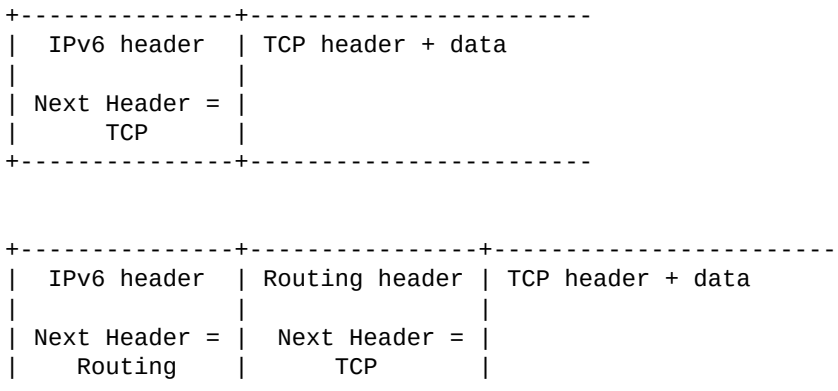
El formato del nuevo encabezado IPv6 se muestra a continuación.

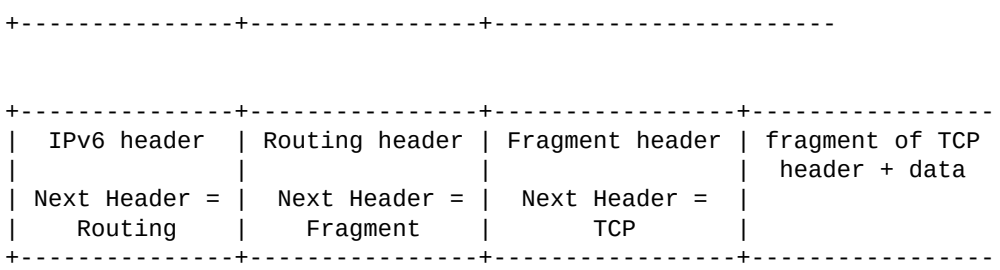


Los diferentes campos que se definen son:

- `Version`: 4 bits para número de versión del Protocolo Internet. En este caso su valor es 6.
- `Traffic Class`: 8 bits para definir las clases de tráfico. Permite que tanto el nodo originador del paquete o los routers intermedios puedan clasificar y distinguir entre distintas clases o prioridades de tráfico IP.
- `Flow Label`: 20 bits para que el origen pueda etiquetar flujos de paquetes que requieran un tratamiento especial por parte de los routers IPv6, ya sea por QoS o servicios de tiempo real.
- `Payload Length`: Entero sin signo de 16 bits. Largo del *payload* IPv6 en bytes, es decir el resto del paquete tras el encabezado incluyendo los diferentes encabezados de extensión.
- `Next Header`: 8 bits para la identificación del tipo de encabezado inmediatamente después del encabezado IPv6. Incluye indicación de encabezados de extensión o de protocolos de capa superior (TCP, UDP).
- `Hop Limit`: Entero sin signo de 8 bits. Se decrementa en 1 por cada nodo que haga el *forwarding* del paquete. Cuando dicho valor alcanza el límite de 0, se descarta el paquete.
- `Source Address`: 128 bits para la dirección de origen.
- `Destination Address`: 128 bits para la dirección de destino.

En IPv6, la información adicional de capa de red se codifica como encabezados de extensión que se ubican entre el encabezado IPv6 y el encabezado de capa superior. Cada tipo de encabezado de extensión se identifica con un valor en el campo `Next Header` visto anteriormente. El siguiente ejemplo busca ilustrar que un paquete IPv6 puede incluir uno, varios o ningún encabezado de extensión y que por lo tanto será necesario un campo `Next Header` en cada caso.





Los diferentes encabezados de extensión que se definen en el estándar son: *Hop by Hop*, *Routing (Tipo 0)*, *Fragment*, *Destination Options*, *Authentication* y *Encapsulating Security Payload*. Salvo el encabezado *Hop by Hop*, ningún otro deberá ser examinado a lo largo del camino recorrido por el paquete hasta alcanzar el nodo destino (identificado por la dirección en el encabezado IPv6). En destino, las funciones de capa de red deberán examinar cada uno de los encabezados de extensión en estricto orden antes de pasar el contenido del paquete a las capas superiores.

Entrar en detalles sobre la funcionalidad de cada una de las opciones excede lo que se pretende presentar en esta sección. Por detalles adicionales, buenas referencias son [8] y [10].

IPv6 impone un requerimiento sobre la MTU mínima en cada enlace de la red pero a diferencia de IPv4 no exige un tiempo de vida máximo para los paquetes. Esto se refleja en la sustitución del campo `Time to Live` de IPv4 por `Hop Limit` en IPv6. Sin embargo, no existen casi aplicaciones en la actualidad donde dicho campo se interprete como un tiempo y por lo tanto en la práctica no habrá diferencias.

6.1.2. Arquitectura de direccionamiento

Como se mencionó inicialmente, la gran ventaja de IPv6 recae en su gigantesco espacio de direcciones. Las nuevas direcciones IPv6 son identificadores de 128 bits para interfaces o conjuntos de interfaces. Todos los aspectos relativos a la arquitectura de direccionamiento se especifican en la RFC 2373.

Existen tres tipos de direcciones:

- **Unicast**: son identificadores para una única interfaz. Un paquete enviado a una dirección unicast es entregado a la interfaz identificada por dicha dirección.
- **Anycast**: son identificadores para un conjunto de interfaces (típicamente pertenecientes a diferentes nodos). Un paquete enviado a una dirección anycast es entregado a **una** de las interfaces identificadas por dicha dirección (a la más cercana del origen, en el sentido del protocolo de ruteo).

- **Multicast**: son identificadores para un conjunto de interfaces (típicamente pertenecientes a diferentes nodos). Un paquete enviado a una dirección multicast es entregado a **todas** las interfaces identificadas por dicha dirección.

Las direcciones broadcast de IPv4 han sido eliminadas.

El modelo de direccionamiento indica que todos los tipos de direcciones IPv6 identifican interfaces y no nodos. Por lo tanto, cualquiera de las direcciones asignadas a las interfaces de un nodo será una dirección unicast válida para identificar al mismo. Otro aspecto importante que se mantiene de IPv4 es que prefijos de subred se asocian a enlaces físicos.

Existen tres formas de representar direcciones IPv6:

1. La forma preferida es `x:x:x:x:x:x:x:x`, donde las `x`'s son los valores en hexadecimal para los 8 campos de 16 bits que componen cada dirección. Un par de ejemplos serían:

   ```
   FEDC:BA98:7654:3210:FEDC:BA98:7654:3210

   1080:0:0:0:8:800:200C:417A
   ```

   Observar cómo no es necesario incluir los ceros de mayor orden en cada campo. Sin embargo, siempre debe haber algún valor especificado en cada campo, salvo que se cumplan las condiciones que se indican a continuación:

2. Debido a la forma natural en que tienden a asignarse las direcciones IPv6, es común que contengan largas cadenas de ceros consecutivos. Para comprimir la notación en estos casos se ha definido una sintaxis especial. El símbolo `::` indica múltiples campos consecutivos de ceros y puede aparecer una sola vez por dirección. Por ejemplo, las siguientes direcciones

   ```
   1080:0:0:0:8:800:200C:417A
   FF01:0:0:0:0:0:0:101
   0:0:0:0:0:0:0:1
   0:0:0:0:0:0:0:0
   ```

   pueden representarse de manera compacta así

   ```
   1080::8:800:200C:417A
   FF01::101
   ::1
   ::
   ```

3. Otra forma alternativa que puede llegar a ser muy conveniente en ambientes mixtos IPv4 e IPv6 es `x:x:x:x:x:x:d.d.d.d` donde las `x`'s representan los seis campos de 16 bits de mayor orden en hexadecimal y las `d`'s representan en decimal, los 4 octetos de menor orden correspondientes a la notación estándar para una dirección IPv4. Ejemplos:

```
0:0:0:0:0:0:13.1.68.3

::FFFF:129.144.52.38
```

Los prefijos se representan de igual manera que en IPv4 utilizando la notación CIDR. Luego de la dirección IPv6 se coloca una `/` seguida de un número decimal que indica cuántos de los bits contiguos y comenzando de la izquierda especifican el prefijo. Un ejemplo:

```
12AB::CD30:0:0:0:0/60
```

Los diferentes tipos de direcciones se identifican mediante secuencias específicas en los bits de mayor orden. Por más detalles, consultar [9].

6.2. Migración a IPv6: ¿qué debe cambiarse?

Antes de comenzar a analizar los cambios a introducir, se consideró importante estudiar en qué se verían afectados los diferentes componentes del sistema al trabajar con IPv6.

En la especificación de las transacciones SIP, no se hace referencia alguna al protocolo de capa de red subyacente. Sin embargo, se presupone el uso de IPv4 ya que SIP es un protocolo para el establecimiento de sesiones multimedia en Internet. Existen varias recomendaciones, publicadas en *Internet Drafts* de la IETF, que indican cómo debería ser el funcionamiento del protocolo SIP en escenarios que contienen clientes y servidores *dual stack*. Uno de los cambios que debe introducirse en el protocolo se da en su procesamiento de las direcciones de red: ahora, en lugar de trabajar únicamente con direcciones que consistan en cuatro números decimales separados por el símbolo "`.`", también deberá incluirse el caso en que las mismas estén definidas entre paréntesis rectos, sean más largas, e incluyan el símbolo "`:`". Más información al respecto puede encontrarse en [13] y [14].

Dado que * mantiene dos canales aislados para cada llamada, no existe ningún inconveniente en que un cliente IPv4 y un cliente IPv6 hablen entre sí, siempre y cuando el servidor * sea *dual stack*.

Debido a esto, desde el punto de vista de la señalización, los usuarios llamante y llamado no necesitan saber cuál es el protocolo de capa de red utilizado por el otro extremo, ya que no hay paquetes intercambiados directamente entre ellos.

En el caso del intercambio de audio, ocurre lo mismo siempre y cuando el audio pase también a través de *, para que éste pueda hacer la "traducción". Esta funcionalidad de * evita que se requiera un nuevo servidor para hacer la traducción, como lo especifican las recomendaciones de la IETF [13] [14]. En este escenario, es imposible intercambiar el flujo RTP directamente entre las puntas. La figura 6.1 ilustra esto.

El principal cambio que introduce IPv6 se da a nivel de la API de *sockets*. Los cambios en la API incluyen una nueva estructura para el almacenamiento de direcciones IPv6, nuevas

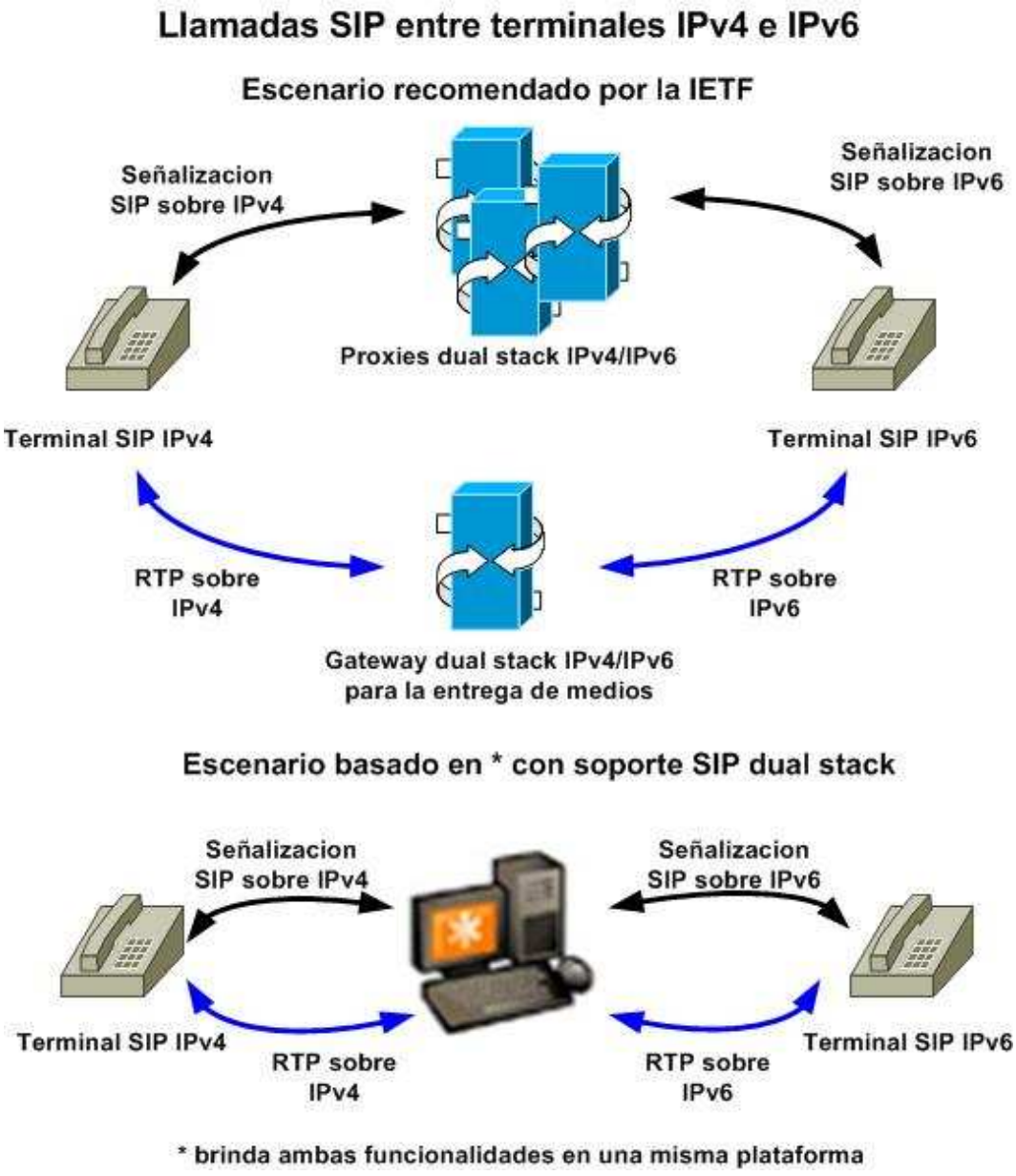


Figura 6.1: Llamadas SIP entre terminales IPv4 e IPv6.

funciones de traducción de direcciones, y algunas nuevas opciones a nivel de *socket*. Estas extensiones son diseñadas para proveer acceso a las funcionalidades básicas de IPv6 (primitivas) requeridas por TCP y UDP, buscando que los cambios sean mínimos, y que se provea una completa compatibilidad con las aplicaciones existentes sobre IPv4. Algunas recomendaciones al respecto son descritas en la RFC 3493 [15]. Dichas recomendaciones no son seguidas al pie de la letra por nuestra implementación, pero son respetadas en su mayoría.

6.3. Punto de partida

Al investigar sobre el tema, se descubrió un *bounty*[2] publicado en el Wiki de * [19] por Bernhard Schmidt, en el cual precisamente se solicitaba la funcionalidad de IPv6 para el canal SIP de *. En este *bounty* se hacía mención a un *patch* desarrollado por Mikael Magnusson que había sido enviado a la lista de desarrolladores de * [24].

Las modificaciones de Magnusson supuestamente eran un primer intento para dar compatibilidad IPv6 para los canales SIP e IAX, para la versión del CVS al momento de haber sido implementadas. Sin embargo, el *patch* era parcial y no funcionaba correctamente; además, su desarrollo había sido abandonado. Fue así que las modificaciones planteadas por Magnusson se convirtieron en el punto de partida para el desarrollo de nuestros propios cambios.

A grandes rasgos, las tareas realizadas en orden cronológico se listan a continuación:

1. Estudio de las modificaciones originales realizadas por Mikael Magnusson (para el CVS de aproximadamente tres meses atrás).
2. Actualización de los cambios e introducción de correcciones, para dar soporte IPv6 en el canal SIP de la última versión estable a la fecha (1.0.8). Pruebas de funcionamiento y publicación del *patch* en la lista de desarrolladores de *.
3. Actualización a la versión estable (1.0.9). Cambios menores y pruebas de funcionamiento.
4. Motivados por los requerimientos para la publicación del *patch* en el *Bugtracker*[3], adaptación del original para que pueda aplicarse a la versión actual del CVS (CVS HEAD, del 22 de Julio de 2005).

6.4. Nueva biblioteca de manejo de *sockets* IPv6

El cambio principal que se introduce para soportar IPv6 en el canal SIP de * consiste en la creación de una nueva estructura `net_sockaddr` para almacenar las direcciones IPv6 (entre otras cosas). Ésta reemplaza a la actual estructura utilizada para IPv4 (`sockaddr_in`) y contiene los siguientes campos:

```
struct net_sockaddr {
    struct sockaddr_storage addr;          /* Espacio para el almacenamiento
                                           de la direccion IPv6 */
    int ai_family;                             /* Familia de direcciones {AF-INET,
                                           AF-INET6} */
    int ai_socktype;                       /* Socket type {SOCK_STREAM,
                                           SOCK_DGRAM} */
```

[2] *Bounty* es una solicitud formal para el desarrollo de cierta funcionalidad requerida por el interesado, a cambio de dinero.

[3] *Bugtracker* es el sistema oficial para que la comunidad de usuarios pueda publicar sus *patches* ya sea para solucionar y/o informar problemas como para agregar nuevas funcionalidades. El sistema es arbitrado por el equipo de desarrolladores oficiales.

```
    int ai_protocol;                        /* Protocolo de capa superior
                                           {IPPROTO_UDP, IPPROTO_TCP}  */
    socklen_t addrlen;                     /* Largo de la sock address storage */
};
```

La definición de esta estructura tiene una gran semejanza con la de las estructuras sugeridas en [15]. Sin embargo, se realizan algunas simplificaciones respecto a éstas, obteniendo un híbrido entre lo utilizado para IPv4 nativo y lo que realmente se sugiere para IPv6, disminuyendo así el número de estructuras involucradas. De hecho, se hace uso de algunas estructuras definidas en `netinet/in.h`; sin ir más lejos, de `sockaddr_storage`.

Para manejar esta estructura, se define una nueva biblioteca de funciones, las cuales se pueden agrupar de la siguiente forma:

- creación y destrucción de la estructura.
- manipulación de estructuras.
- lectura y asignación de valores de los campos de la estructura.
- conversión de dirección y puerto a diferentes formatos.
- manejo de máscaras.
- manejo de *sockets*.

La mayoría de las funciones ya existían para la estructura IPv4 pero debieron ser modificadas para poder trabajar con la nueva estructura IPv6, más compleja.

La definición de la estructura `net_sockaddr`, en conjunto con toda su batería de funciones, son introducidas en el código de * a través de la inclusión de los nuevos archivos fuente `net.c` y `net.h`.

En las siguientes secciones se describirán las principales características de las funciones pertenecientes a cada grupo, citando algunos ejemplos.

6.4.1. Creación y destrucción de estructuras

Toda estructura debe ser creada antes de poder trabajar con ella. La función que hace esto para la estructura IPv6 es `net_addr_new`. Lo que hace es reservar un espacio de memoria para la estructura, e inicializar todos sus campos en cero (lo cual se hace a través de la función `net_addr_clr` descrita más adelante). Su código se muestra a continuación:

```
net_sockaddr *net_addr_new(void) {
    net_sockaddr *sa = malloc(sizeof(net_sockaddr));
    net_addr_clr(sa);
    return sa;
```

```
}
```

La destrucción de la estructura se hace mediante la función `net_addr_delete`. Lo que hace es liberar el espacio de memoria asignado a la estructura (mediante la función `free`). Su código es el siguiente:

```
int net_addr_delete(net_sockaddr *sa) {
    free(sa);
    return 0;
}
```

También existe la posibilidad de reservar memoria para la estructura, sin inicializarla. Esto se hace a través de la función `net_addr_alloca`.

6.4.2. Manipulación de estructuras

Por manipulación de estructuras se entiende a toda aquella funcionalidad que incluya copia, comparación e inicialización de la estructura completa. No se abarcan aquellas funciones que actúan sobre los campos particulares, ya que éstas se describirán más adelante.

Un primer ejemplo claro de esto es la función `net_addr_cpy`, que efectúa la copia de una estructura origen a una estructura destino, copiando los espacios de memoria de una a otra:

```
int net_addr_cpy(net_sockaddr *dst, net_sockaddr *src) {
    memcpy(dst, src, sizeof(net_sockaddr));
    return 0;
}
```

Otra función perteneciente a esta categoría es la ya mencionada `net_addr_clr`, que inicializa el espacio de memoria de la estructura en 0.

También debe mencionarse la función `net_addr_cmp`, usada para la comparación de estructuras.

6.4.3. Lectura y asignación de valores de los campos de la estructura

Para esta arquitectura *dual stack*, es muy importante contar con una función que obtenga la familia de la dirección con la que se está trabajando (IPv4 o IPv6). Gran parte del procesamiento de la estructura dependerá de saber qué tipo de dirección contiene, ya que en muchos casos el tratamiento de la misma varía dependiendo de si es IPv4 o IPv6. La función que implementa esto es `get_ss_family`, que toma como parámetro una estructura del tipo `net_sockaddr` y devuelve la constante `AF_INET` si la familia de la dirección es IPv4, o `AF_INET6` si la familia es IPv6. Estas constantes son definidas en la API de *sockets*.

Dos funciones que también se encuentran en este grupo son `net_addr_setport` y `net_addr_getport`. La primera le asigna un valor entero al puerto, y la segunda devuelve el valor del mismo. La función `net_addr_cpyport` es un ejemplo claro del uso de las otras dos:

```
int net_addr_cpyport(net_sockaddr *dst, net_sockaddr *src) {
   int port = net_addr_getport(src);

   if (port < 0) {
       return port;
   }

   return net_addr_setport(dst, port);
}
```

Esta función primero guarda el valor del puerto de la estructura origen en la variable `port` y luego le asigna ese mismo valor al puerto de la estructura destino. Si el puerto de origen es menor que 0 (o sea, es un puerto no válido), se sale de la función.

Finalmente, se presenta también la función `net_addr_ishostnull`, que verifica si la dirección de red es nula. Es muy ilustrativa ya que muestra en un caso puntual, cómo deben tratarse de manera diferente las estructuras que almacenan la direcciones de red IPv4 e IPv6.

```
int net_addr_ishostnull(net_sockaddr *sa)
{
        int res = 0;

        switch(get_ss_family(sa)) {
        case AF_INET: {
                struct sockaddr_in *in =
                        (struct sockaddr_in *)&sa->addr;
                res = in->sin_addr.s_addr == 0;
                break;
        }
        case AF_INET6: {
                struct sockaddr_in6 *in =
                        (struct sockaddr_in6 *)&sa->addr;
                int i;
                res = 1;

                for (i=0; i<4; i++) {
                        if (in->sin6_addr.s6_addr32[i] != 0) {
                                res = 0;
                                break;
                        }
                }
                break;
        }
        default:
                res = -1;
```

```
        }

        return res;
}
```

La función se basa en un `case` que separa dos posibilidades según la dirección sea IPv4 o IPv6 (para determinarlo, se usa la función `get_ss_family` descrita más arriba).

Para el caso de IPv6, se almacena la dirección en el puntero local `in` mediante un *cast* al tipo `sockaddr_in6`. Esta estructura es la recomendada por la IETF para el almacenamiento de direcciones IPv6 y mediante un bucle se verifica que las cuatro palabras de 32 bits que componen la dirección sean nulas. La función retorna 1 si la dirección es nula, 0 en caso contrario, y -1 ante un error.

6.4.4. Conversión de dirección y puerto a diferentes formatos

Se han definido nuevas implementaciones independientes del protocolo para la traducción de *hosts* a direcciones de *sockets*, y recíprocamente. Estas nuevas funciones son `getaddrinfo` y `getnameinfo` respectivamente[4]. La especificación detallada de las mismas se encuentra en [15].

Como ejemplo de utilización de estas nuevas funciones dentro de `net.c`, consideraremos la función `net_aston` que realiza la traducción de nombres de *host* a direcciones:

```
int net_aston(const char *cp, const char *serv, net_sockaddr *ia)
{
        int error;
        struct addrinfo *hinfo = NULL;
        struct addrinfo hints;
        char *buf = ast_strdupa(cp);
        char *ptr = buf;
        char *host;
        int len = strlen(buf);

        if (ptr[0] == '[' && ptr[len-1] == ']') {
            host = ptr + 1;
            ptr[len-1] = '\0';
        } else {
            host = ptr;
        }

        memset(&hints, 0, sizeof(hints));
        hints.ai_family = ia->ai_family;
        hints.ai_socktype = ia->ai_socktype;
        hints.ai_protocol = ia->ai_protocol;

        error = getaddrinfo(host, serv, &hints, &hinfo);
```

[4] Para acceder a dichas funciones, se deben incluir las librerías `sys/socket.h` y `netdb.h`.

```
        if (error) {
                return 0;
        }

        memcpy(&ia->addr, hinfo->ai_addr, hinfo->ai_addrlen);
        ia->addrlen = hinfo->ai_addrlen;
        ia->ai_family = hinfo->ai_family;
        ia->ai_socktype = hinfo->ai_socktype;
        ia->ai_protocol = hinfo->ai_protocol;

        return 1;
}
```

Esta función se basa principalmente en la función `getaddrinfo`. En primer lugar, se verifica si el argumento que se pasa como *host* incluye paréntesis rectos y en caso afirmativo se los elimina.

Luego se actualizan los campos de la variable `hints` del tipo `addrinfo`. A través de esta variable es que se le pasa la información de familia IP, tipo de *socket* y protocolo de capa superior a la función `getaddrinfo`, de manera que pueda procesar la consulta adecuadamente.

Finalmente, tras realizar la consulta, se actualizan los valores de la estructura `net_sockaddr` que se pasa como parámetro para ser modificada. La función devuelve 0 en caso de error, o 1 en caso contrario.

El ejemplo inverso de esta función, que es prácticamente un *wrapper* de `getnameinfo`, es `net_ntoa`.

Otras funciones que se encuentran en este grupo son:

- `net_ntourl`, que devuelve la dirección y puerto en formato URL en un *string*.
- `net_ntostr`, que devuelve el par dirección:puerto en un *string*.
- `net_ntos`, que devuelve el puerto en un *string*.

6.4.5. Manejo de máscaras

Se definen tres funciones para el trabajo con máscaras:

- `net_addr_initmask`, que convierte la máscara en notación de barra (compatible con IPv4 e IPv6) en una secuencia de 1s y 0s, de acuerdo a la familia.
- `net_addr_checkmask`, que verifica si un *host* pertenece a una red dada IPv4 o IPv6. Para eso, toma como parámetros las estructuras `net_sockaddr` del *host*, de la máscara y de la red, le aplica la máscara a la dirección del *host* y la compara con la de la red.
- `net_addr_andmask`, que toma como parámetros las estructuras `net_sockaddr` del *host* y de la máscara, y le aplica la máscara a la dirección del *host*.

6.4.6. Manejo de *sockets*

Las funciones clásicas para el manejo de sockets UDP (`socket`, `bind`, `recvfrom` y `sendto`) han sido redefinidas para lograr la compatibilidad con la estructura `net_sockaddr`. Las nuevas funciones son: `net_socket`, `net_bind`, `net_recvfrom` y `net_sendto`. Estas funciones son básicamente un *wrapper* de las definidas en la API de *sockets*. Un ejemplo claro es el de `net_socket`:

```
int net_socket(net_sockaddr *sa) {
   return socket(sa->ai_family, sa->ai_socktype, sa->ai_protocol);
}
```

La gran diferencia radica en que ahora se llama a `socket` pasando como parámetro la familia contenida en el campo `family` de la `net_sockaddr`, mientras que antes en * todas las creaciones de `socket` se especificaban con la familia `AF_INET` fija.

6.5. Modificaciones al código de *

Otro punto que debió tenerse en cuenta fue el alcance de las nuevas funciones y estructura definidas, a nivel de los archivos fuente de *. En esta etapa se debió analizar en qué archivos serían utilizadas dichas funciones. El siguiente es el listado de archivos fuente que fueron modificados teniendo en cuenta esto, considerando además que únicamente se busca dar soporte IPv6 al canal SIP:

- `chan_sip.c`: implementación del canal SIP de *.
- `rtp.c` y `rtp.h`: implementación del canal RTP de audio.
- `acl.c` y `acl.h`: implementación del manejo de diversas funcionalidades de red para *.
- `netsock.c` y `netsock.h`: surgen en la última versión de *, contienen algunas funciones antes implementadas dentro de `acl.c`.
- `dnsmgr.c` y `dnsmgr.h`: surgen en la última versión de *, contienen funciones relativas al manejo de DNS.
- `Makefile` general de *.

Partiendo de la base de que el soporte IPv6 sería brindado únicamente para el canal SIP, por ser éste el objeto de nuestro estudio, no se modificó ninguno de los demás protocolos que, al igual que SIP, utilizan RTP para el envío del audio (Skinny y MGCP). Por lo tanto, los módulos de los canales Skinny y MGCP son deshabilitados (mediante modificaciones realizadas al `Makefile`), con el objetivo de evitar errores de compilación.

Un esquema general de los cambios introducidos se muestra en la figura 6.2. En ella se indican los archivos fuente modificados debido a la inclusión de las nuevas funciones implementadas en `net.c`.

Los archivos `dnsmgr.c`, `dnsmgr.h`, `netsock.c` y `netsock.h`, que se indican en punteado, sólo fueron modificados al adaptar los cambios al CVS HEAD.

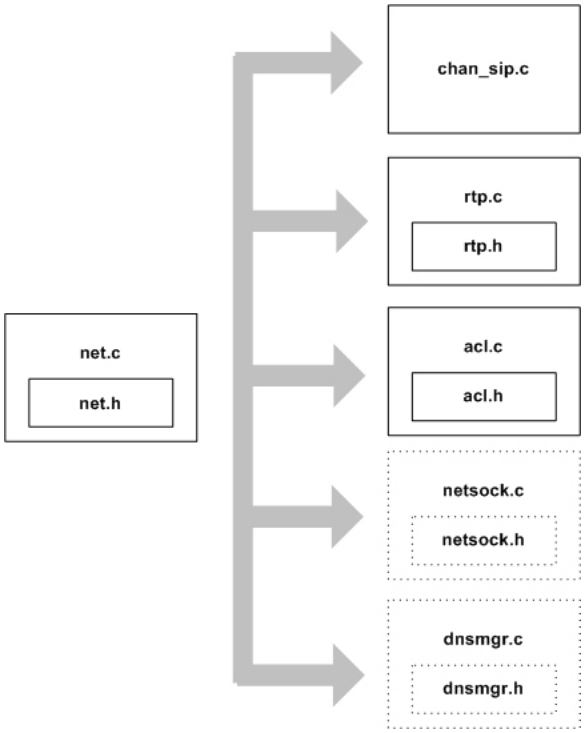


Figura 6.2: Esquema de los cambios realizados. Los archivos en punteado se modificaron sólo en la versión del CVS HEAD.

Debido a la arquitectura modular de *, se puede afirmar con certeza que los archivos mencionados serán los **únicos** que deberán ser modificados, y que no habrá funciones aisladas en otros archivos fuente relacionadas con la vieja estructura.

6.5.1. Ejemplos de cambios realizados

Es evidente que muchos de los cambios a implementar serán sistemáticos; por ejemplo:

- donde aparecen, las estructuras `sockaddr_in` deberán ser reemplazadas por `net_sockaddr`. Debe realizarse también la reserva del espacio de memoria para esta estructura, lo cual se hace a través de la función `net_addr_alloca`.
- deberán ser cambiados todos los campos de `sockaddr_in` que son referenciados en el código, por los de la nueva estructura.

- los buffers para el almacenamiento temporal de direcciones deberán ser de mayor tamaño. Antes los mismos eran de tamaño `INET_ADDRSTRLEN` (32 bits), y ahora deberán ser de tamaño `NET_ADDRSTRLEN` (constante igual a `INET6_ADDRSTRLEN`, 128 bits).

También se deberán sustituir aquellas funciones que son afectadas por el agregado de soporte IPv6; las funciones a usar en su lugar son las definidas en la nueva biblioteca, descrita en la sección anterior.

Un ejemplo de esto es el reemplazo de la función `ast_inet_ntoa`. La misma agrupa un conjunto de funcionalidades que en la nueva biblioteca fueron separadas y asignadas a las funciones `net_ntoa`, `net_ntostr` y `net_ntourl`. Dependiendo de para qué se esté usando la función `ast_inet_ntoa`, la misma es sustituida por:

- `net_ntoa` cuando se quiere obtener la resolución de DNS.
- `net_ntostr` cuando se quiere desplegar el par dirección IP:puerto en consola.
- `net_ntourl` cuando se quieren definir los campos `Contact` y `Via` del paquete SIP (un claro ejemplo de esto se da dentro de las funciones `build_contact` y `build_via`).

A través del uso de nuevas funciones del tipo `get` y `set`, se eliminan las referencias directas a los campos de las estructuras. Una clara ventaja de esto es la mayor prolijidad que adquiere el código. Como ejemplo, se puede citar la nueva función `net_addr_setport`. A continuación se muestra cómo dicha función reemplaza una asignación directa al campo que contiene el puerto en la estructura `r` (del tipo `sockaddr_in` o `net_sockaddr`, según el caso):

```
/* Asignación directa, antes de los cambios realizados */
r->sa.sin_port = htons(portno);

/* Uso de la función setport, luego de la implementación de los cambios */
net_addr_setport(r->sa, portno);
```

6.6. El escenario de pruebas

En esta sección se describirán algunas de las pruebas realizadas sobre *, para comprobar su correcto funcionamiento luego de la implementación de las diversas modificaciones.

Primero se describe la red que se armó a tales efectos, y luego cómo fueron realizadas dichas pruebas.

6.6.1. La red de pruebas

Para realizar las pruebas de funcionamiento de * sobre IPv6 fue necesario montar una red de pruebas con dicho protocolo de capa de red. En una primera instancia, las pruebas fueron realizadas sin resolución de DNS. La topología que se planteó en este caso era muy sencilla: un *switch*, el servidor * y dos computadoras con Linux para correr los UA (*softphone* Linphone).

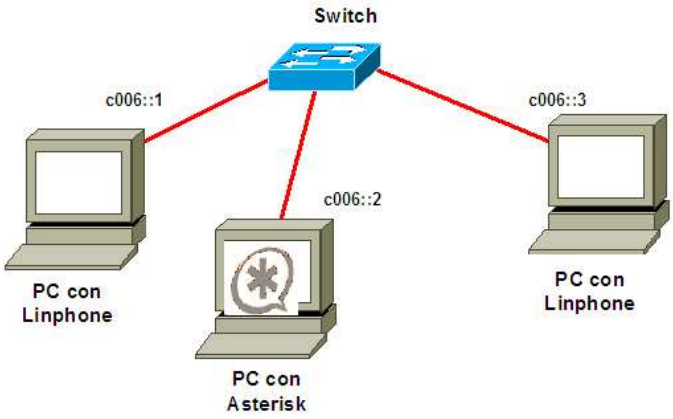


Figura 6.3: Red sencilla para pruebas de llamadas sobre IPv6.

Se utilizó la red con prefijo `c006::/16`, asignando las direcciones `c006::1` y `c006::3` a los UA, y `c006::2` al *. Un esquema de la topología de dicha red se muestra en la figura 6.3. La configuración de las direcciones para las interfaces se logra ejecutando el siguiente comando como *root*:

```
ifconfig <interfaz> add <dirección IPv6>/<Máscara>
```

Por ejemplo, para la máquina corriendo el *:

```
Asterisk# ifconfig eth0 add c006::2/16
```

Para comprobar la correcta conectividad entre los nodos, se utilizó el comando `ping6`, que prueba la conectividad mediante el envío de paquetes ICMP en IPv6. Dado que además todos los Linux tenían corriendo un servidor SSH, se realizaron pruebas de conectividad con conexiones SSH, utilizando el parámetro `-6` que realiza este tipo de conexiones sobre IPv6.

Luego de realizar pruebas con la red IPv6 pura, se le configuró una dirección IPv4 a *, de manera que el mismo pasó a ser *dual stack*. Así se pasó a tener una red en la que se encontraban corriendo IPv6 e IPv4 al mismo tiempo. A ésta se le sumó un nuevo UA IPv4 (X-Lite sobre Windows) para realizar las pruebas de comunicación en un ambiente dual.

En una etapa posterior, se configuró el servidor de nombres (DNS) BIND para atender consultas IPv6, y así poder probar las nuevas funciones relacionadas `getaddrinfo` y `getnameinfo`. La configuración del BIND no difiere mucho de la configuración típica de un DNS en una red IPv4 pura; los archivos de configuración utilizados se incluyen en el Apéndice E. [10] fue una buena referencia para llevar a cabo estas configuraciones.

El cambio más importante que se introduce en el servicio de nombres es que ahora, para obtener una dirección IPv6 relacionada a una etiqueta, se debe realizar una consulta solicitando el registro AAAA (*quad-A*) en lugar de la consulta típica que se realiza en IPv4 por el registro A. Por esta razón, en el archivo de configuración de la zona en la que se encuentran

los *hosts* IPv6, se deben agregar las entradas adecuadas para poder responder las consultas *quad-A*.

Dado que el servidor BIND actualmente no soporta funcionamiento *dual stack* (es decir, atiende consultas o bien IPv4 o bien IPv6, pero no ambas simultáneamente), se debió iniciar el servidor de nombres con el parámetro `named -6`. En esta expresión, `-6` le indica al servidor que debe escuchar consultas de *hosts* IPv6. Además, como otro elemento destacable, en el archivo de configuración `/etc/named.conf` se debió agregar la línea `listen-on-v6 {any;};` dentro de `options`.

Finalmente, se armó el Escenario 3 ya visto en la sección 5.3 para comprobar que el NAT (implementado para IPv4) seguía funcionando correctamente luego de realizadas las modificaciones.

6.6.2. Pruebas efectuadas

En esta sección se describen las pruebas realizadas para verificar el funcionamiento de las extensiones implementadas.

Pruebas sobre IPv4

En primer lugar, luego de haber efectuado las modificaciones y compilado * sin errores, se apuntó a verificar que el funcionamiento de * para un escenario puramente IPv4 seguía siendo correcto, y si todas las funcionalidades con las que contaba el sistema antes de los cambios se habían mantenido.

Las primeras pruebas resultaron ciertamente desalentadoras: cuando se intentaba realizar una llamada sobre IPv4, el servidor se caía dando como salida *broken pipe* o *segmentation fault*. Hubo entonces todo un proceso de depuración, donde se hicieron seguimientos de las llamadas (usando las funciones de registro en consola de *, a través del método `ast_log`), para poder ubicar los problemas y luego solucionarlos.

En la gran mayoría de estos casos, los problemas se debían a la falta de inicialización de punteros a estructuras `net_sockaddr`. Este tipo de problema se resolvió agregando la sentencia `net_addr_alloca(&sin);`, donde `sin` es un puntero a `net_sockaddr`.

Pruebas en IPv6 y dual stack

Una vez que se logró un funcionamiento estable de * para redes IPv4, se probó la nueva funcionalidad en IPv6. Con esto surgió otro elemento para nada menor: contar con un cliente SIP IPv6 que funcionase correctamente. Se encontraron únicamente dos opciones, ambas para

Linux: los *softphones* Kphone y Linphone.

El proyecto de desarrollo del Kphone se encuentra dividido en dos ramas: una rama principal dedicada exclusivamente a IPv4, y otra para brindar soporte IPv6. Dado que esta última se encontraba sin actividad desde hacía casi un año, y que surgieron diversos problemas al compilar la aplicación, se decidió de manera primaria dejar de lado esta posibilidad e intentar con el Linphone.

Por su parte, Linphone es un proyecto que sigue en desarrollo con soporte para ambas versiones del protocolo IP, y tiene listas de correo en funcionamiento que permiten la evacuación de dudas. La instalación requería bibliotecas y paquetes adicionales, pero el proceso se pudo completar con éxito.

Luego de montada la red IPv6 con * y dos PCs corriendo el Ubuntu Linux con el Linphone como UA, se comenzaron las pruebas en la plataforma IPv6.

Para el correcto funcionamiento de * en *dual stack*, se debió modificar la línea `bindaddr=0.0.0.0` por `bindaddr=[::0]` en el archivo de configuración `sip.conf`.

Una vez que el sistema estuvo a punto, se realizó una serie de pruebas para corroborar si todo funcionaba como lo esperado. Siguiendo los pasos de configuración para el Linphone descritos en la sección 5.2, se logró que dicho UA se registre en IPv6:

```
asterisk*CLI>
    -- Registered SIP '100' at c006::3;5060 expires 900
    -- Saved useragent "Linphone-1.0.1/eXosip" for peer 100
asterisk*CLI>
```

```
asterisk*CLI> sip show peers
Name/username    Host            Dyn Nat ACL Mask             Port     Status
105/105          (Unspecified)    D   N      255.255.255.255  -1       Unmonitored
104/104          (Unspecified)    D          255.255.255.255  -1       Unmonitored
103/103          (Unspecified)    D          255.255.255.255  -1       Unmonitored
102/102          192.168.0.15     D          255.255.255.255  5060     Unmonitored
101/101          (Unspecified)    D   N      255.255.255.255  -1       Unmonitored
100/100          c006::3          D          255.255.255.255  5060     Unmonitored
asterisk*CLI>
```

Inicialmente se llamó al número de correo de voz (*voicemail*); lo cual implicaba una comunicación únicamente entre el UA y el *. Esta conexión fue realizada con éxito, tanto a nivel de señalización SIP como en lo que respecta al intercambio de medios RTP.

La siguiente prueba consistió en realizar una llamada entre dos UAs IPv6. Esta vez la conexión no fue exitosa, ya que cuando se realizaba la llamada, el UA receptor timbraba pero el UA llamante recibía el mensaje de usuario no disponible (*unavailable*). Si el usuario llamado decidía

atender, evidentemente no escuchaba nada del otro lado, pues la interconexión de canales con el usuario llamante nunca se producía. Con el objetivo de detectar el origen de este nuevo problema, se realizaron pruebas adicionales incluyendo UAs SIP IPv4 y teléfonos analógicos conectados a módulos FXS. Por esta vía se logró detectar que el problema se daba únicamente cuando se trataba de llamar a un UA IPv6, independientemente de quién iniciara la llamada. De hecho, cuando un UA IPv6 iniciaba una llamada a otro tipo de UA (no IPv6), ésta se completaba satisfactoriamente.

Para solucionar este problema, se hizo un estudio del flujo de la llamada en * siguiendo el código en `chan_sip.c`, el cual resultó infructuoso dado que no se encontró en qué punto * estaba fallando. Como siguiente paso, se procedió a analizar el flujo de intercambio de mensajes SIP mediante capturas con el Ethereal. De este estudio se desprendió que el Linphone tenía un *bug* trabajando en IPv6, inexistente al trabajar sobre IPv4. El *bug* consistía en que no enviaba el mensaje `200 OK` para indicar sus capacidades de intercambios de medios a través del SDP y finalizar el establecimiento de la llamada.

Tras realizar una consulta a la lista de usuarios de Linphone, se pudo averiguar que efectivamente el *bug* ya había sido reportado, y que Steve Ayer estaba completando su trabajo en un *patch* que solucionaba el problema. Al día siguiente, dicho *patch* fue enviado a la lista y luego de ser aplicado, obtuvimos un correcto funcionamiento de Linphone.

En la figura 6.4 se muestra el ejemplo de un flujo de mensajes SIP para el establecimiento de llamada entre dos Linphones, uno de ellos en IPv4 y el otro en IPv6. Es interesante notar que *, en operación *dual stack*, se ve como dos entidades de red aisladas. La vinculación de esta dos entidades no se da a nivel de red, sino mediante las funciones centrales de la PBX.

También se muestra un fragmento del INVITE enviado por * al Linphone IPv6 en el proceso de establecimiento del canal de salida para la llamada (mensaje número 117). Se puede observar cómo se transportan las direcciones IPv6 en los diferentes encabezados, y también en el *payload* SDP. Notar el uso de paréntesis rectos según la recomendación, para distinguir los ":" propios de la dirección, del ":" utilizado como símbolo de separación entre la dirección y el puerto UDP.

```
INVITE sip:100@[c006::3]:5060 SIP/2.0
Via: SIP/2.0/UDP [c006::2]:5060;branch=z9hG4bK2d81a0b9
From: "102" <sip:102@[c006::2]>;tag=as6de5fd01
To: <sip:100@[c006::3]:5060>
Contact: <sip:102@[c006::2]>
Call-ID: 1b04eebe0efd63236f1e686947a1cd4a@c006::2
CSeq: 102 INVITE
User-Agent: Asterisk PBX
Date: Mon, 08 Aug 2005 20:44:11 GMT
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER
Content-Type: application/sdp
Content-Length: 251
Message body
```

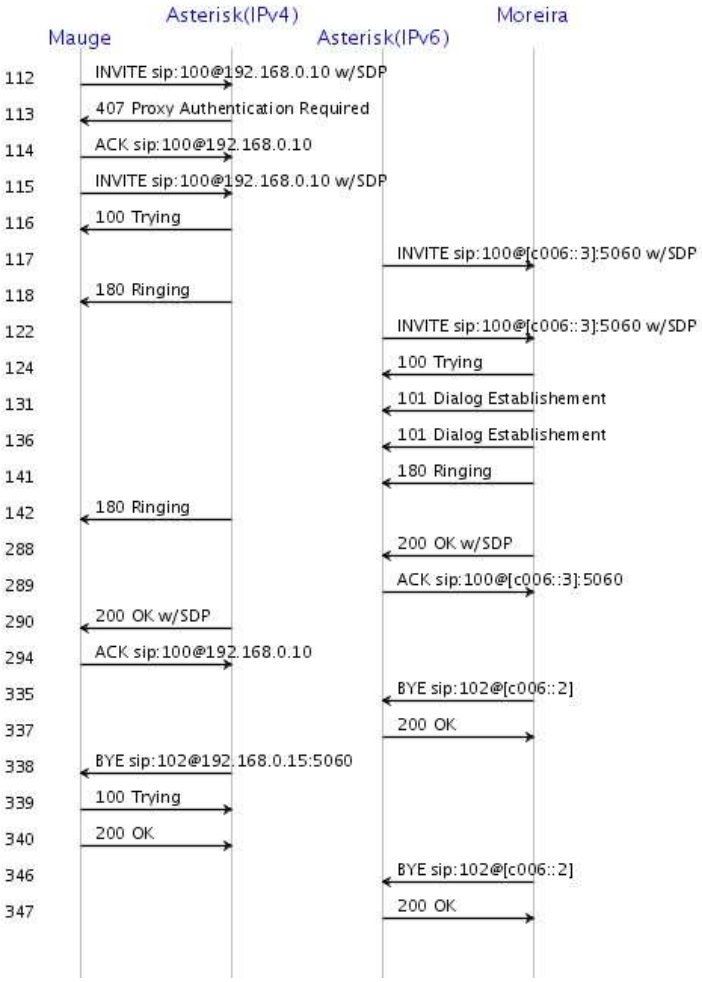


Figura 6.4: Flujo de mensajes SIP en una llamada IPv4-IPv6.

```
Session Description Protocol Version (v): 0
        Owner/Creator, Session Id (o): root 1774 1774 IN IP6 c006::2
        Session Name (s): session
        Connection Information (c): IN IP6 c006::2
...
```

A partir de este punto se continuaron las pruebas, verificando el correcto funcionamiento del sistema ante todas las combinaciones posibles de tipos de clientes origen y destino. Nuevamente, la ventajosa arquitectura de * que conecta canales aislados de entrada y salida, brindaba la tranquilidad de que al poder realizar llamadas entrantes y salientes a un cliente SIP IPv6, las otras combinaciones también funcionarían. En particular, para los cambios aplicados al CVS HEAD se hicieron pruebas adicionales con funcionalidad avanzada de llamadas: transferencias directas y asistidas, conferencias con múltiples usuarios, etc. El funcionamiento

fue satisfactorio en todos los casos.

Pruebas en escenarios NAT IPv4/IPv6

En este punto, se procedió a corroborar el correcto funcionamiento del NAT. Recordar que antes de implementadas las modificaciones, el comportamiento de SIP frente a diversos escenarios NAT había resultado satisfactorio[5].

Primero que nada se hicieron pruebas con * en la red pública y los clientes detrás del NAT, y las llamadas se realizaron con éxito.

En una segunda instancia, se situó el * detrás del NAT, y los clientes fuera. El escenario montado en este caso fue similar al escenario 2 de la sección 5.3. Los *port forwardings* en el *router*, y las variables `externip` y `localnet` se configuraron de manera coherente con lo ya visto.

En este caso, se observó que los clientes no podían registrarse. Esto nos llevó a un proceso de búsqueda del origen del problema, que consistió en varios pasos que se describen a continuación:

1. Detectar si el problema fue introducido por nuestras modificaciones para dar soporte IPv6, o por la última versión del CVS. Para ello, se instaló dicha versión del CVS sin modificaciones, y se observó que el funcionamiento en el mismo escenario era satisfactorio. Por lo tanto, se concluyó que el problema había sido introducido por los cambios realizados.

2. Se analizaron los mensajes SIP intercambiados entre los UAs y *, mediante capturas Ethereal. Aquí se descubrió que en los mensajes SIP generados por * hacia los clientes en la red pública, los encabezados SIP `Contact` y `Via`, y el campo `Connection Information` del SDP no estaban siendo actualizados con la dirección `externip`. Se observó que estos campos mantenían la dirección IP privada.

3. Se concluyó que el problema debía buscarse en la función que implementaba dicha sustitución de direcciones IP en escenarios NAT. Dicha función es `ast_sip_ouraddrfor`, y se encuentra en `chan_sip.c`.

La función `ast_sip_ouraddrfor` recibe un puntero a la dirección IP de * en el campo `us`, y un puntero a la dirección IP del cliente en el campo `them`. La lógica de funcionamiento de `ast_sip_ouraddrfor` se ilustra en la figura 6.5.

Luego, la condición que se debe verificar para copiar `externip` en la dirección IP de * es:

```
localaddr && externip && ast_apply_ha(localaddr,theirs)
```

[5] Por mayores referencias, ver la sección 5.3.

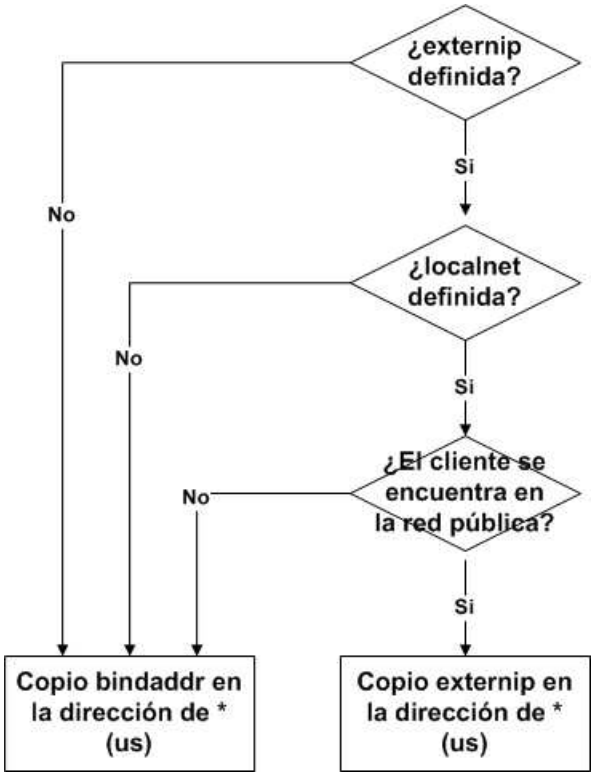


Figura 6.5: Lógica de ejecución de la función ast_sip_ouraddrfor (soporte para SIP detrás de NAT).

En este nuevo escenario, donde conviven IPv6 y NAT para redes IPv4, se da el caso de que como la red IPv6 siempre es distinta de la `localnet` (red IPv4 privada), en caso de recibir mensajes SIP de un cliente IPv6 * siempre tratará de cambiar su dirección equivocadamente. Esto se debe a que la condición mostrada arriba se cumple, y por lo tanto * interpreta que debe hacer NAT.

Por eso, se debió agregar una condición adicional antes de realizar la copia. Esta condición verifica que la dirección del mensaje SIP que llega sea o bien IPv4 nativa o bien IPv4 mapeada en IPv6. Entonces, luego de las modificaciones, la condición que debe verificarse es:

```
localaddr && externip && ast_apply_ha(localaddr,theirs) &&
    ((net_addr_getfamily(them)==AF_INET) || net_addr_isv4mapped(them))
```

Esto solucionó parte del problema, pero aún así el mecanismo NAT no funcionaba ya que en los mensajes SIP de * seguía apareciendo su dirección privada.

Para efectuar la copia de direcciones se usa la función `net_addr_cpyhost` en lugar de `net_addr_cpy`. Esta última es la función usada comúnmente para la copia de estructuras `net_sockaddr` en *. ¿Por qué aquí se usa `net_addr_cpyhost`? Porque los campos de

conexión del *socket* no están definidos, ya que en este caso la estructura `externip` sólo se usa para guardar una dirección.

Mediante un monitoreo de las variables antes y después de la copia se observó que el error era interno a la función `net_addr_cpyhost`. Por lo tanto, se hicieron modificaciones a nivel de esta función.

Se descubrió que el problema consistía en que la función presuponía que sus argumentos (dirección origen de la copia, y dirección destino de la copia) pertenecían a la misma familia; si esto no era así, se salía de la función indicando error, sin efectuar la copia:

```
if (dstfamily != get_ss_family(src)) {
       return -1;
}
```

Para este caso (sustitución de direcciones cuando se está usando NAT) el destino de la copia (`us`) es una dirección recién inicializada, por lo que su familia en principio no está especificada (está definida como `PF_UNSPEC`), mientras que el origen de la copia (`externip`) sí tiene su familia definida (que es `AF_INET`). En consecuencia, la función así implementada nunca iba a poder efectuar la copia correctamente.

Esto fue modificado por el siguiente código, donde en caso de que origen y destino tengan familias diferentes, se copia la familia del origen en la familia del destino:

```
if (dstfamily != get_ss_family(src)) {
    ast_log(LOG_WARNING,"Different Families.\n");
    switch(srcfamily) {
        case AF_INET:{
            ast_log(LOG_WARNING,"SRC Family is AF_INET.\n");
            net_addr_setfamily(dst, AF_INET);
            break;
        }
        case AF_INET6:{
            ast_log(LOG_WARNING,"SRC Family is AF_INET6.\n");
            net_addr_setfamily(dst, AF_INET6);
            break;
        }
        default:{
        ast_log(LOG_WARNING,"Family error SRC.\n");
        return -1;
        }
    }
}
```

Aún con este cambio, la función no realizaba la copia correctamente. Lo que se hizo en este punto fue modificar la parte del código que asignaba la dirección origen a la dirección destino. El código original era:

```
struct sockaddr_in *ia_dst =
                (struct sockaddr_in *)&dst->addr;
struct sockaddr_in *ia_src =
                (struct sockaddr_in *)&src->addr;
ia_dst->sin_addr.s_addr = ia_src->sin_addr.s_addr;
```

El código modificado resultó:

```
dst->addr = src->addr;
```

Donde `dst` es la dirección destino de la copia, y `src` es la dirección origen de la copia. En este caso, `dst` es `us`, y `src` es `externip`.

Después de implementadas todas estas modificaciones, se resolvió correctamente el problema planteado. Los clientes IPv4 en la red pública pudieron registrarse correctamente en *, tal como ocurría en escenarios IPv4 puros, previo al agregado de la extensión para IPv6.

A modo de conclusión, se puede afirmar que luego de diversas etapas de depuración todas las llamadas fueron exitosas, tanto del punto de vista de su establecimiento, como de su desarrollo y liberación. Puede razonablemente concluirse que las modificaciones implementadas resuelven en forma correcta el uso de * como PBX en escenarios IPv4 puros, IPv6 puros, y *dual stack*, cumpliendo así con el objetivo planteado en esta etapa.

7. DISEÑO DE UN SISTEMA BASADO EN *

7.1. Introducción

El presente capítulo muestra la implementación de un sistema de telefonía basado en * desde cero. Esto implica: evaluación de los requerimientos del sistema; elección del hardware a utilizar; definición de los archivos de configuración de *; y desarrollo de aplicaciones en C para implementar posibles funcionalidades requeridas por el sistema y actualmente no ofrecidas por *.

Para esta etapa se decidió armar un sistema que fuese capaz de emular el funcionamiento de la central telefónica existente en el Instituto de Ingeniería Eléctrica (I.I.E.) de la Facultad de Ingeniería. Se eligió este sistema por varias razones:

- es un sistema medianamente grande, lo cual representaba un desafío en cuanto a la configuración de la central.
- existe información sobre la central actualmente disponible en el servidor del I.I.E. que deja muchos puntos definidos acerca de qué funciones debe cumplir la misma y cómo debe hacerlo.
- todos los integrantes del grupo han tenido contacto con el I.I.E., por lo que se conoce la central de cerca y se ha podido consultar a docentes que han trabajado con ella y que tienen más experiencia en su uso.

A continuación se mostrarán y explicarán las distintas etapas transcurridas en esta parte del proyecto.

7.2. Requerimientos del sistema

Tal como se mencionó en la Introducción, antes de comenzar a desarrollar el sistema fue necesario estudiarlo y definir cuáles eran sus requerimientos.

No sólo se tuvieron en cuenta los requerimientos desde el punto de vista del funcionamiento en sí, sino que también se contemplaron los elementos físicos que sería necesario considerar si se quisiera hoy en día sustituir la central existente.

Desde este último punto de vista, se observó que cada sala del I.I.E. cuenta con uno o más teléfonos, cada uno identificado por un número de interno. Luego, el número de interno para

contactar a un docente del I.I.E. coincide con el número de interno del teléfono que se encuentra en la sala donde éste trabaja (en el caso de que se trate de una sala con más de un teléfono, entonces se elige uno de ellos).

En cuanto a líneas externas, actualmente el I.I.E. cuenta con cinco líneas urbanas integradas en una línea colectiva, conectadas directamente a ANTEL, y una línea de fax, que entra al servidor ampere separada del resto. Pensando en sustituir la central por un sistema basado en *, se debe entonces tener en cuenta este requisito, el cual contemplará únicamente a las líneas urbanas. La razón por la que se decidió no trabajar con el fax a nivel de * es que, por más que * sí es capaz de manejar líneas de fax, el sistema que está implementado actualmente no forma parte de la central, funciona correctamente y se quiso mantener. Dicho sistema consiste en que ampere atiende las posibles llamadas entrantes, las procesa, y almacena las imágenes que pueden ser impresas en la secretaría (siendo también accesibles desde la web interna).

La central actual brinda la siguiente funcionalidad para el manejo de llamadas entrantes. Cuando uno llama al I.I.E., es atendido por una grabación, que da la opción de marcar el interno directamente, si uno lo conoce; luego informa el interno de la operadora; y por último da la opción de listar un directorio ordenado por apellidos. El flujo que sigue la llamada dentro del menú IVR (*Interactive Voice Response*) puede verse en la figura 7.1.

7.3. Elección de hardware

A la hora de elegir el hardware a instalar en el sistema, se tuvo en cuenta tanto la funcionalidad brindada por los dispositivos disponibles en el mercado, así como su precio. Se realizó un balance costo-beneficio llegando a una posible solución basada en una arquitectura VoIP.

Se consideraron dos posibilidades: sustituir los internos analógicos existentes por teléfonos SIP; o bien la incorporación al sistema de adaptadores de teléfonos analógicos (ATAs), los cuales permiten la conexión de teléfonos analógicos a una solución de voz sobre IP como la que se pensaba imponer.

Luego de una comparación de precios, se decidió la opción de los adaptadores, en particular el Sipura SPA 2100[1]. Su costo unitario es de aproximadamente USD 100, y su principal ventaja radica en que al tener dos puertos Ethernet, no es necesario realizar un recableado significativo, sino que se aprovecha el cableado Ethernet existente. Esto es posible gracias a que se pueden desconectar las PCs actualmente integradas a la red; se conectan esos cables a uno de los puertos de los Sipuras; y finalmente se utilizan los segundos puertos de los Sipuras (cuya existencia es la principal ventaja de estos dispositivos frente a otros de prestaciones similares) para conectar las PCs que quedaron en principio desconectadas de la red. Como puede notarse, el cableado que hay que realizar en sí es prácticamente insignificante, ya que implica

[1] Por información técnica sobre el equipo, consultar [28].

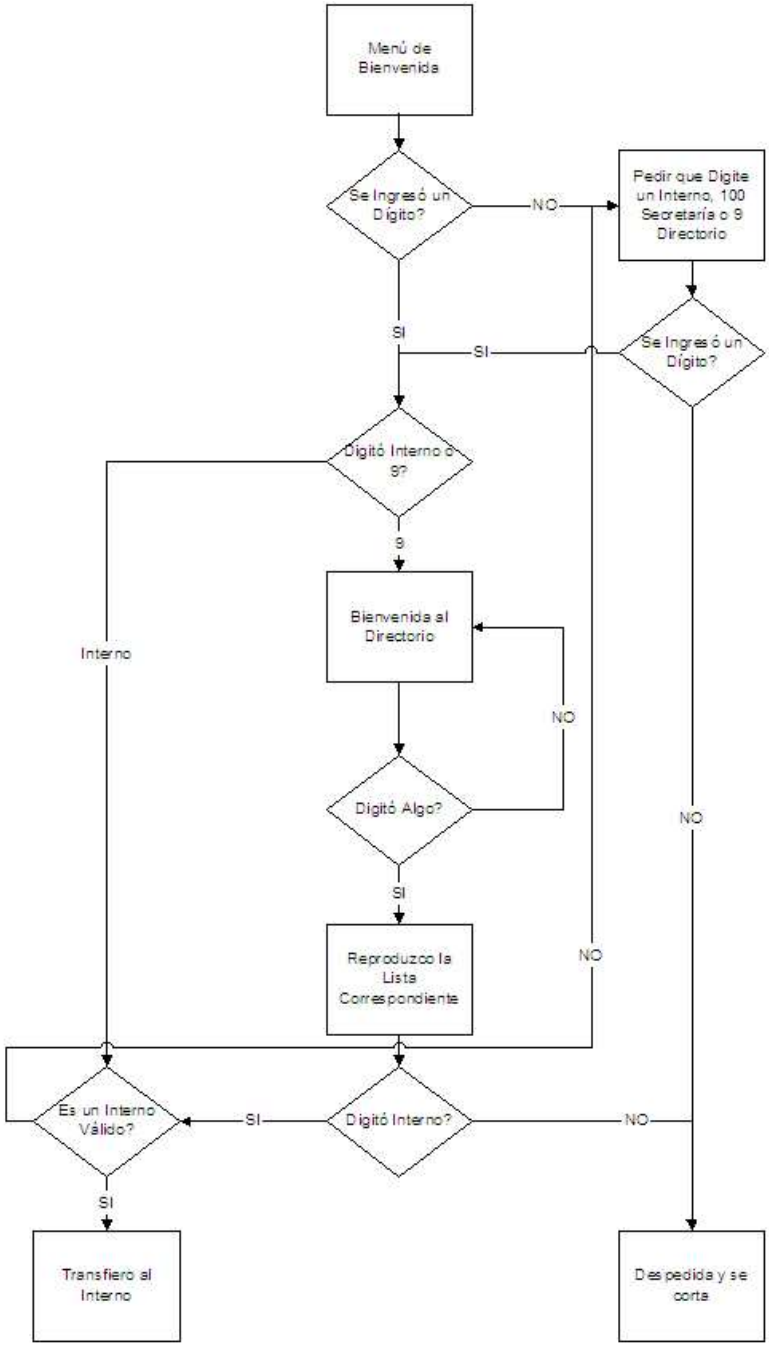


Figura 7.1: Flujo de una llamada al ingresar a la central telefónica del I.I.E..

únicamente la tirada entre los Sipuras y PCs situadas en la misma sala.

Teniendo en cuenta la cantidad de teléfonos analógicos que deben integrarse, y la disposición

de los mismos en las salas, la cantidad de Sipuras que habría que instalar es de 16, ya que cada Sipura cuenta con dos interfases telefónicas. La siguiente figura, figura 7.2, muestra la distribución pensada para los Sipuras y los teléfonos (identificados por sus internos).

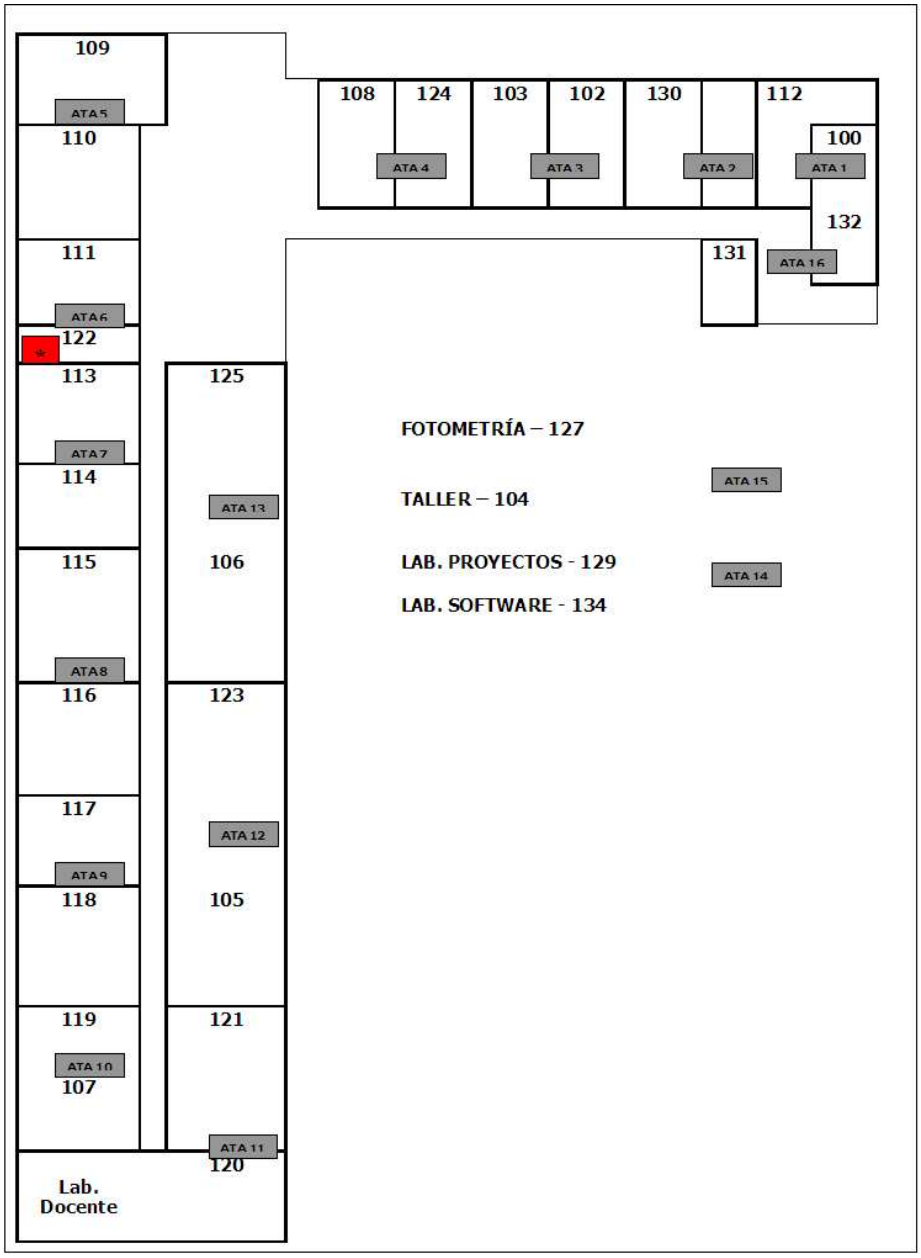


Figura 7.2: Esquema de distribución de Sipuras y teléfonos en el I.I.E..

Lamentablemente, los teléfonos analógicos con los que hoy cuenta el I.I.E., de cuatro hilos, son incompatibles con las salidas de los Sipuras, de dos hilos. Además, todas las teclas de

Dispositivo	Cant.	Costo Unit. USD	Costo Total USD
Sipura SPA 2100	16	100,00	1.600,00
Teléfono analógico	32	10,00	320,00
TDM13B	2	329,00	658,00
PC	1	500,00	500,00
UPS	1	150,00	150,00
		Total USD	**3.228,00**

Tabla 7.1: Costo de hardware para sistema en el I.I.E..

función de los teléfonos actuales serían inutilizadas. En consecuencia, se pensó en sustituirlos por teléfonos de dos hilos, más sencillos (es decir, sin funciones especiales como aquellas con las que cuentan los teléfonos actuales). El costo aproximado de cada teléfono es de USD 10, y se necesitarían 32 teléfonos para armar el sistema.

Las líneas externas son cubiertas a través de la instalación de tarjetas FXO Digium Wildcard modelo TDM13B, en el servidor *. Cada tarjeta posee un puerto FXS y tres puertos FXO, y por lo tanto el suministro incluye dos[2] (con lo que se tienen dos puertos FXS y seis puertos FXO, uno más que los existentes actualmente[3]). Su costo unitario es de USD 329.

La gran ventaja de contar con puertos FXS consiste en que en el caso de un corte de luz, se tienen líneas telefónicas disponibles, solamente agregando una pequeña UPS de respaldo para el servidor. Si no se contara con estas líneas, y se quisiera seguir disponiendo del servicio telefónico ante un corte de luz, entonces debería instalarse una UPS con mayor capacidad. Esta UPS debería operar como respaldo del servidor y todos los ATAs a instalar, lo cual resulta una opción menos conveniente, básicamente desde el punto de vista de los costos.

Finalmente, para cumplir las funciones de servidor se pensó en una PC Pentium IV, de 2.8 GHz, 512 MB de RAM DDR, y 80 Gb de disco. Su precio unitario es de aproximadamente USD 500. Para respaldo de la PC, también se incluye una UPS de potencia 800 VA (540 Watt), la cual podrá brindar respaldo durante 20 minutos. Su costo unitario ronda los USD 150.

A modo de resumen, la tabla 7.1 muestra el hardware que se suministraría, acompañado de su costo unitario y total.

[2] Esta solución no pudo ser validada en la práctica, ya que no se contó con los elementos necesarios. Es importante aclarar esto ya que en la lista de Digium [25] se han reportado problemas cuando se usa más de una tarjeta.

[3] En la sección 7.4.7 se muestra que en realidad este puerto sobrante puede utilizarse para el portero eléctrico.

7.4. Configuración de *

A nivel de *, la lógica del menú IVR en el sistema fue definida dentro del archivo `extensions.conf`; las extensiones SIP de los usuarios del sistema se definieron en el archivo `sip.conf`; las extensiones Zaptel en `zapata.conf`; y finalmente, todas las casillas de correo fueron creadas en `voicemail.conf`. También se adaptó el archivo `features.conf` para mantener los mismos números de aparcado y levantado de llamadas existentes actualmente, y se definió el archivo `meetme.conf` para la funcionalidad de conferencias multi-usuario[4].

Se decidió usar el codec G.729 (8 kbps) para los internos SIP a instalar.

En las próximas secciones se describirá la arquitectura general del sistema a nivel de su lógica de manejo de llamadas, así como sus elementos constitutivos básicos, que han permitido brindar las funcionalidades requeridas.

7.4.1. Arquitectura general: los contextos

Las principales entidades que se reconocen dentro del sistema son:

- el menú de bienvenida.
- el directorio por apellidos.
- los internos SIP.
- las líneas urbanas.
- los internos analógicos.

Todos estos elementos deben interactuar en el *dialplan* de forma de respetar la lógica de manejo de llamadas existente en el sistema actual. Para cumplir con este objetivo, dentro del *dialplan* se definieron los siguientes contextos:

- `[menu-principal]`, que implementa el IVR (menú de bienvenida).
- `[directorio]`, que contiene la lógica de funcionamiento del directorio organizado por apellidos.
- `[desde-sip]`, donde se manejan todos los internos SIP.
- `[salientes]`, donde se manejan todas las llamadas salientes del sistema (a través de las líneas urbanas).
- `[emergencia]`, que incluye los internos analógicos para realizar llamadas en caso de un corte de luz.

[4] Los archivos de configuración completos se encuentran en el Apéndice D.

- `[llamadas_aparcadas]`, que contiene los internos donde se aparcan las llamadas.

En la figura 7.3 se muestra un esquema de los contextos definidos y su interacción en el *dialplan* para implementar la funcionalidad requerida.

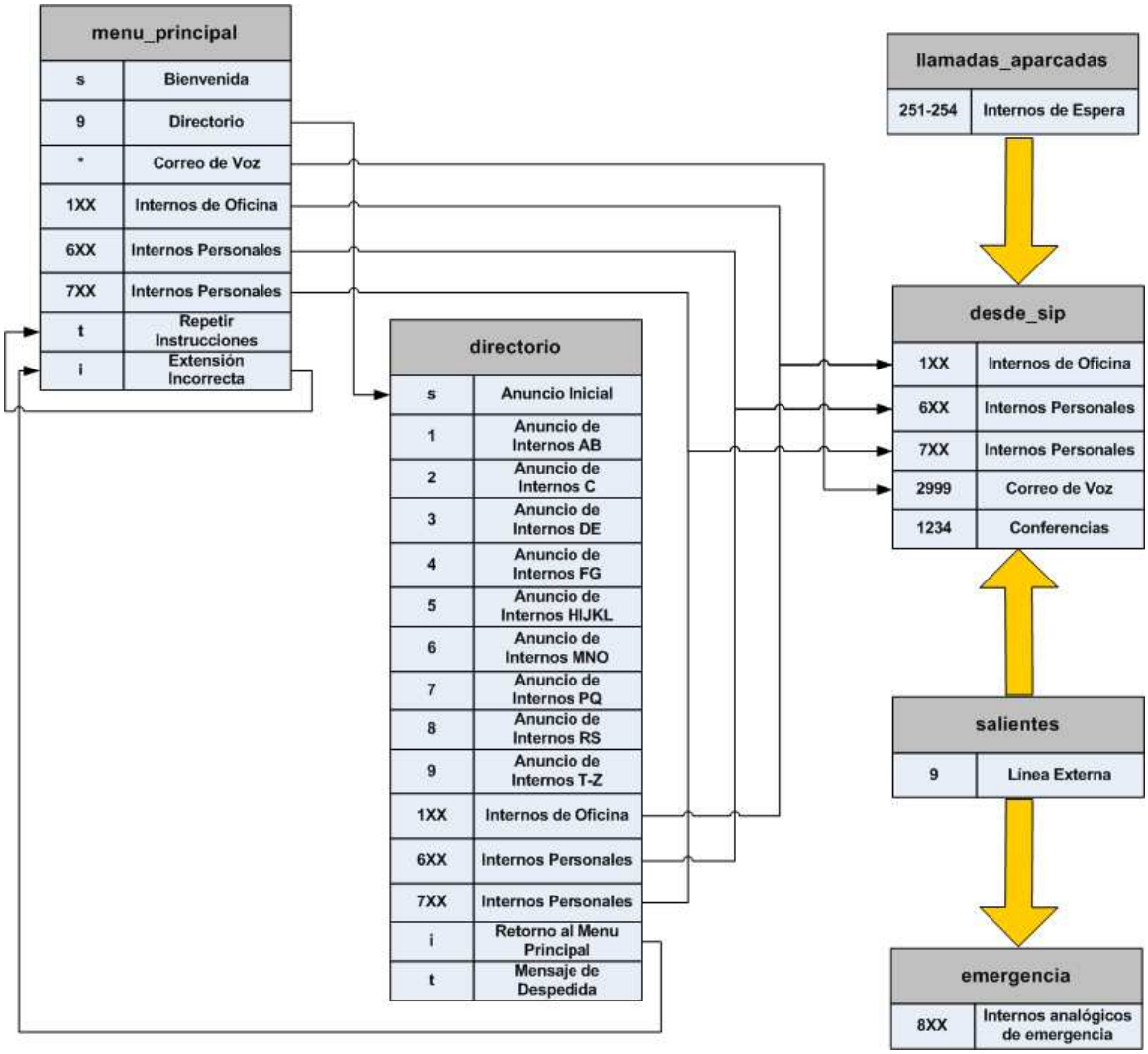


Figura 7.3: Diagrama de interacción entre contextos.

Las flechas de color negro indican saltos entre extensiones (implementados por la aplicación `GoTo` en el *dialplan*). La extensión origen y la extensión destino pueden o no pertenecer al mismo contexto, y se puede elegir a qué prioridad ir dentro de la extensión destino. Por ejemplo, desde las extensiones 1XX, 6XX y 7XX del contexto `menu-principal` se salta a las extensiones 1XX, 6XX y 7XX respectivamente del contexto `desde-sip`; estas últimas extensiones llaman a los respectivos internos SIP. El siguiente fragmento del archivo `extensions.conf`

ilustra lo necesario para realizar dicho salto. Notar que en la variable `${EXTEN}` se guarda el número de la extensión actual.

```
[menu-principal]

...

exten =>_[167]XX,1,Goto(desde-sip,${EXTEN},1)

...
```

Las flechas de color anaranjado indican la inclusión de contextos. Por ejemplo, el contexto `salientes` es incluido dentro de `desde-sip` y `emergencia`. Esto permite realizar llamadas salientes, a través de las líneas urbanas, desde los internos SIP y desde los internos analógicos de emergencia. El siguiente fragmento del archivo `extensions.conf` ilustra lo necesario para realizar dichas inclusiones.

```
...

[emergencia]

;Teléfonos de emergencia

include => salientes

...

[desde-sip]

;Internos SIP

include => salientes

...
```

7.4.2. Menú de bienvenida

El menú de bienvenida, definido en el contexto `menu-principal`, es el accedido por cualquier llamada que entra al sistema. El fragmento del *dialplan* que contiene este contexto se muestra a continuación.

```
[menu-principal]

;Menú de bienvenida

exten =>s,1,ResponseTimeout(5)
exten =>s,2,Background(bienvenida)
exten =>s,3,Background(instrucciones)
```

```
exten =>t,1,Background(instrucciones)
exten =>t,2,Background(silence/5)
exten =>t,3,Playback(despedida)
exten =>t,4,Hangup

exten =>i,1,Playback(ext_incorrecta)
exten =>i,2,Goto(menu-principal,t,1)

exten =>9,1,Goto(directorio,s,1)
exten =>*,1,Goto(desde-sip,2999,1)
exten =>_[167]XX,1,Goto(desde-sip,${EXTEN},1)
```

Una vez que una llamada entrante arriba a este contexto, se reproduce un mensaje de bienvenida, y luego las instrucciones a seguir:

- Marcar un interno.
- Acceder a secretaría (interno 100).
- Acceder al directorio por apellidos.

Esto se define en la extensión `s` (*start*) del contexto. La reproducción de archivos de sonido, que se grabaron en formato `.gsm`, se hace mediante la aplicación `Background` de `*`, que permite reproducir un archivo de sonido y al mismo tiempo capturar los tonos (DTMF) ingresados por el usuario.

Llegado este punto, surgen tres posibilidades:

- que el usuario no marque ninguna extensión luego de transcurridos 5 segundos (este tiempo es definido mediante el llamado a la aplicación `ResponseTimeOut`). En este caso, el flujo se deriva a la extensión `t` (*timeout*) del mismo contexto. En este punto, se reproducen nuevamente las instrucciones, se espera en silencio durante 5 segundos, y finalmente se corta la comunicación si el usuario no ingresó ninguna extensión, luego de reproducir un mensaje de despedida (prioridades 1 a 4 de la extensión `t`).
- que el usuario marque una extensión incorrecta (no definida en el contexto). En este caso, el flujo se deriva a la extensión `i` (*invalid*), donde se reproduce un mensaje acorde, y luego se salta a la prioridad 1 de la extensión `t`. En este punto la situación es la misma que la descrita en el ítem anterior.
- que el usuario marque una extensión correcta (definida en el contexto). En este caso, el flujo se deriva a otro contexto: el del directorio si se marca 9, el del correo de voz si se marca `*`, o el del interno en particular si se digita correctamente su número (en los dos últimos casos el contexto al que se va es `desde-sip`).

7.4.3. Internos personales y redirección

Tal como se mencionó anteriormente, hoy por hoy cada docente del I.I.E. puede ser contactado a través del interno de la sala donde trabaja. El hecho de utilizar una arquitectura de VoIP basada en SIP, sumado a la posibilidad de tener *softphones* disponibles en las PCs del I.I.E., motiva la idea de que cada docente pueda tener su propio número de interno.

Una de las principales ventajas de este mecanismo consiste en que cualquier docente podrá registrarse desde cualquier máquina dentro del I.I.E., que tenga un *softphone* corriendo; tan sólo necesita registrarse en el servidor. Se observa una vez más que la movilidad que proporciona SIP es un aspecto muy útil; cuando el usuario se registra, * actualiza su ubicación y le envía correctamente sus llamadas.

Para cada docente, el número de interno personal debe estar vinculado de alguna forma con su interno de oficina, para que en el caso de que no esté utilizando su *softphone*, la llamada a su interno personal pueda ser redireccionada a su oficina.

Los internos personales fueron asignados de manera única para cada docente, en el rango comprendido entre 600 y 698[5].

Luego, se definió un mapeo entre el interno personal de cada docente, y su interno actual, determinado por la sala en que trabaja. Este mapeo se cargó automáticamente en la base de datos de *, por un mecanismo desarrollado especialmente para esta aplicación, que se describirá más adelante en esta misma sección.

El funcionamiento pensado para contactar a un docente a través de su interno personal con la posibilidad de redirección se muestra en el diagrama de flujo de la figura 7.4.

Como se ve en el diagrama, un primer paso importante es obtener de la base de datos de * el interno de oficina relacionado con el interno personal que se marcó.

En la sección que sigue, se describirá la aplicación diseñada, `loadDB`, para guardar dicho mapeo en la base de datos de *, y finalmente se mostrará la implementación de la redirección a nivel del *dialplan*.

Desarrollo de aplicación LoadDB

En el proceso de desarrollo de este sistema, se vio que implementar la relación entre internos personales e internos por sala, para casi 100 docentes, no era para nada práctico de llevar a cabo si tenían que ingresarse línea por línea.

A esto se le suma el hecho de que si algún día hubiera que hacer tareas de mantenimiento en el sistema, se debería trabajar con las líneas de la base de datos de * una por una, lo cual

[5] En realidad se reservó el rango entre 600 y 799 para tener la posibilidad de asignar hasta 200 internos.

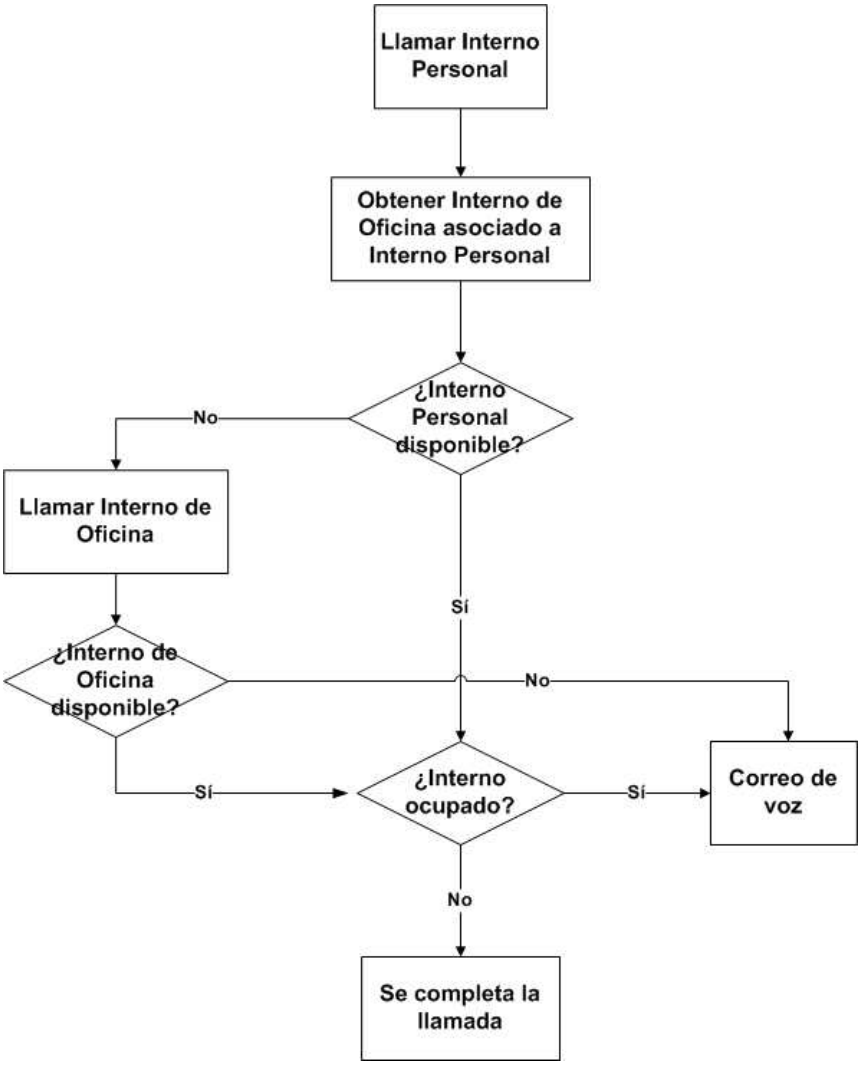


Figura 7.4: Manejo de llamada a un interno SIP con redirección.

resultaría sumamente engorroso.

En consecuencia, se decidió escribir una aplicación que cargue un archivo guardado en formato `.csv` (archivo de valores separados por coma), y vuelque su contenido en la base de datos de *. A esta aplicación se la llamó `loadDB` (*Load DataBase*, o *Cargar Base de Datos*).

Para que la aplicación funcione correctamente deben cargarse los siguientes datos en campos consecutivos del archivo con formato `.csv` (eligiendo como separador el símbolo `;`):

Apellido, Nombre; Interno de Oficina; Interno Personal

Cuando se ejecuta la aplicación, se define un puntero al archivo que contiene la base de datos[6], mediante la sentencia:

```
static char *file ="/etc/asterisk/files/BaseDatos.csv";
```

Si el archivo no se encuentra, entonces no se carga la base de datos y la aplicación devuelve 0 (y no otro valor distinto para que * no se caiga) desplegando un mensaje de error. Esto se hace en la sentencia:

```
    if ((TextFile = fopen(file,"r")) == NULL){
        ast_log(LOG_WARNING,"Unable to open file: %s\n",file);
        return 0;
    }
```

Si por el contrario, el archivo sí se encuentra, se comienza a leer línea por línea con el comando `get_line`; esto se hace hasta que no existen más líneas a leer. La implementación es la siguiente:

```
while((c = getline(&line,&line_size,TextFile)) != -1) {
...
}
```

Luego se cargan los campos del archivo (a través de la función `strtok`), verificando que ninguno de los campos sea vacío. La variable `name` contendrá el nombre del docente; la variable `office` contendrá su interno de oficina; y la variable `ext` contendrá su interno personal:

```
        name = strtok(line,";");
        if (!(name==NULL)){
            office = strtok(NULL,";");
            if (!(office==NULL)){
                ext = strtok(NULL,";\0");
                if (!(ext==NULL)){
                    \* Todos los campos son distintos de NULL *\
                    code = ast_db_put("cidname",ext,name);
                    code = ast_db_put("redirect",ext,office);
                }
```

Si todos los campos están definidos (es decir, ninguna de las variables es `NULL`), se almacenan en la base de datos interna de *, a través de la función `ast_db_put`. Si por el contrario alguno de los campos que se quiso leer no estuviese definido, se sale de la aplicación desplegando un mensaje de error en la consola de * (el mensaje que se muestra es `Formato Incorrecto de Archivo`). Esto se logra a través del siguiente código:

[6] La ubicación del archivo podría definirse en un archivo de configuración. Esto se haría de la forma ya vista en la sección 3.4.1 mediante el uso de la función `ast_load_config`.

```
else {
    fclose(TextFile);
    ast_log(LOG_WARNING,"Formato Incorrecto de Archivo\n");
    return 0;
    }
```

Por último, una vez que terminó de leerse el archivo, se lo cierra, a través de la siguiente sentencia:

```
fclose(TextFile);
```

Si todo funcionó correctamente, se sale de la aplicación retornando 0.

Una vez finalizada la ejecución, queda guardado el vínculo entre el nombre del docente y su interno personal bajo la familia `cidname`, mientras que el vínculo entre el interno personal y el interno por sala se guarda bajo la familia `redirect`. A continuación se presenta un fragmento de la salida del comando `show database` de la CLI de *, donde se muestra para Federico Morales, cómo se almacena su número de interno personal, 661, y el de su oficina para redirección, 115.

```
asterisk*CLI> database show
...
/cidname/661
                                    : MORALES, FEDERICO
/redirect/661
                                   : 115
...
```

Esta aplicación se carga en el momento en que se inicializa el *, ya que dentro de `app_loaddb.c` se registró la aplicación (mediante el método `ast_register_application`) dentro de la función `load_module` del código.

Para que la aplicación sea ejecutada, es necesario hacer un *reload* del módulo a través del CLI de *. Esto se hizo así porque la aplicación se corre una única vez, y no se quiso forzarla a que se ejecutara cada vez que * se inicializara. Para esto, la rutina de la aplicación se escribió dentro de la función `reload_module` de `app_loaddb.c`.

Como esta aplicación normalmente será ejecutada desde el CLI, no tiene sentido definirla en el *requester* dado que éste no puede ser invocado desde el CLI, sólo desde el *dialplan*.

Es inmediato notar que únicamente con editar el archivo `.csv` se pueden hacer cambios en los números de interno asociados a cada uno de los docentes del I.I.E., lo cual hoy en día, con el esquema de central telefónica existente, no es tan sencillo.

El código completo de esta aplicación puede leerse en el Apéndice F.

Implementación de redirección en el dialplan

En esta sección se explicará formalmente cómo se implementa la redirección que ya fue introducida en la figura 7.4. A continuación se muestra el fragmento del *dialplan* que brinda esta funcionalidad.

```
;Extensiones Personales

;Obtengo el interno de oficina asociado al interno personal
;El valor se devuelve en la variable INTERNO
;La búsqueda se hace por la clave redirect/${EXTEN}
;donde ${EXTEN} tiene almacenado el valor del interno personal

exten => _[67]XX,1,DBget(INTERNO=redirect/${EXTEN})

;Llamo al interno personal
;Si da Unavailable, salto a la prioridad 3
;Si da Busy, salto a la prioridad 103

exten => _[67]XX,2,Dial(SIP/${EXTEN},20,Tt)

;Llamo al interno de oficina
;Si da Unavailable, salto a la prioridad 4
;Si da Busy, salto a la prioridad 104

exten => _[67]XX,3,Dial(SIP/${INTERNO},20,Tt)


exten => _[67]XX,4,Voicemail(u${EXTEN})
exten => _[67]XX,5,Hangup
exten => _[67]XX,103,Goto(104)
exten => _[67]XX,104,Voicemail(b${EXTEN})
exten => _[67]XX,105,Hangup
```

Notar que se hace uso de la aplicación `DBGet`, que permite extraer datos de la base de datos a partir de una clave que es pasada como parámetro.

La situación de indisponibilidad (*Unavailable*) del interno personal se va a dar si el docente no se registró desde un *softphone*.

7.4.4. Correo de voz

De acuerdo a nuestra configuración, el correo de voz puede ser accedido mediante la extensión 2999. Esto se logra en el *dialplan* al definir la aplicación `VoicemailMain` como primera prioridad para dicha extensión. Esto se ilustra a continuación.

```
;Correo de voz
```

```
exten => 2999,1,VoicemailMain(${CALLERIDNUM})
```

Las casillas de correo se definen en el archivo `voicemail.conf`. Todas ellas fueron definidas por defecto con clave 1111, pero tanto la clave como el mensaje de voz que se reproduce al acceder a la casilla son configurables por el usuario. Un ejemplo de configuración de una casilla de correo se muestra a continuación.

```
[local]

;
;Formato: numero de casilla => password,nombre,mail para enviar
;mensaje adjunto
;

661 => 1111,Federico Morales,fmorales@fing.edu.uy
```

La gran ventaja del manejo de correo de voz de * es que es capaz de enviar un mail a una dirección de correo configurada en el archivo `voicemail.conf`, propia de cada usuario (en este caso se utilizó la dirección de correo del dominio `fing.edu.uy` para cada docente). El mail contiene adjunto el archivo de voz correspondiente a cada mensaje que fue dejado en la casilla. Esto evidentemente es de suma utilidad para aquellos usuarios del sistema que suelen chequear el mail desde fuera del I.I.E..

7.4.5. Internos analógicos

Los internos correspondientes a los dos teléfonos conectados a las interfaces FXS de las placas Digium, serán usados en caso de emergencia (corte de luz, como ya se mencionó). Se les asignarán los números 800 y 801.

Para la configuración de estos internos, que utilizan los módulos FXS de las tarjetas TDM13B, se deben definir los archivos `/etc/zaptel.conf` y `/etc/asterisk/zapata.conf`. En el primero de ellos se configuran los parámetros de las tarjetas instaladas. Para el hardware elegido para este sistema, la configuración se muestra a continuación. Notar que para cada una de las tarjetas, el primero de los cuatro módulos es el FXS, los restantes son FXO.

```
;
; Este es el archivo de configuración zaptel.conf para
; el servidor asterisk.fing.edu.uy
;

;Utilizar tonos de indicación en formato EE.UU.
loadzone = us

;Utilizar este tipo de tonos por defecto
```

```
defaultzone=us

;El primer módulo de la primera tarjeta es FXS
fxoks=1

;Los tres siguientes módulos de la primera tarjeta son FXO
fxsks=2-4

;El primer módulo de la segunda tarjeta es FXS
fxoks=5

;Los tres siguientes módulos de la segunda tarjeta son FXO
fxsks=6-8
```

Dado que los módulos FXS se conectan a dispositivos que esperan señalización FXO, en esos casos * debe configurarse para generar dicho tipo de señalización. Es por eso que se define como `fxoks` a los módulos FXS. Análogamente, los módulos FXO se configuran como `fxsks`.

La configuración de los canales Zaptel asociados a los módulos FXS y FXO se realiza en el archivo `zapata.conf`. Para los internos de emergencia se configuró lo siguiente:

```
;
;Módulos FXS
;

;Tipo de señalización (FXO para módulos FXS)
signalling=fxo_ks

;Habilitar cancelación de eco
echocancel=yes
echocancelwhenbridged=yes
echotraining=400

;No modificar el callerid
callerid=asreceived

;Grupo al cual pertenecen estos módulos
group=1

;Contexto a donde enviar sus llamadas
context=emergencia

;Ubicación del módulo FXS en la primera placa
channel=>1

;Ubicación del módulo FXS en la segunda placa
channel=>5
```

Como elementos destacables se tienen:

- el tipo de señalización, que será FXO.

- el grupo al cual pertenecen dichos canales (brinda la posibilidad de llamar al grupo y que suene el primero disponible de ese grupo).
- contexto al cual pertenecerán las llamadas provenientes de estos módulos (en este caso, es el contexto `emergencia`).
- cancelación de eco.
- mapeo entre el canal Zaptel y la ubicación del módulo hardware (la relación se efectúa mediante `channel => X` en este archivo, y `fxoks = X` en `zaptel.conf`).

7.4.6. Líneas urbanas

De manera similar a los canales FXS, los canales FXO (correspondientes a las líneas urbanas) se configuran a través de los archivos `zapata.conf` y `zaptel.conf`.

La forma de configurarlos es idéntica a la de los FXS, y los parámetros de configuración muy similares. La principal diferencia consiste en que en este caso se debe señalizar FXS. Esto implica que los módulos FXO serán identificados como `fxsks`. A continuación se muestra el fragmento de `zapata.conf` donde se configuran estos canales.

```
;
;Módulos FXO
;

;Tipo de señalización (FXS para módulos FXO)
signalling=fxs_ks

;Grupo al cual pertenecen estos módulos
group=2

;Contexto a donde enviar sus llamadas
context=menu-principal

;Ubicación de módulos FXO en la primera placa
channel=>2-4

;Ubicación de módulos FXO en la segunda placa
channel=>6-8
```

De acuerdo a esta configuración, las **llamadas entrantes** por las líneas urbanas pertenecen al contexto `menu-principal`. Esto se hizo así dado que en este contexto se define la lógica IVR del menú de bienvenida.

Las **llamadas salientes** del I.I.E. se manejan en el contexto `salientes`. Para poder acceder a una línea externa, primero que nada debe discarse 9. Esto se implementa mediante la sentencia `ignorepat => 9` que le indica a * que mantenga el tono de discado aún luego de

discado el dígito 9.

Se contemplan dos tipos de llamadas: llamadas locales, y llamadas a teléfonos celulares. Los números correspondientes a las primeras serán de 7 dígitos, y no comenzarán por 0 o 1. Los números correspondientes a las segundas serán de 9 dígitos, y comenzarán por 09. La configuración del contexto `[salientes]` en el *dialplan* es la siguiente.

```
[salientes]

;Mantener el tono de línea luego de discar 9
ignorepat =>9

;Salida a teléfono fijo

exten => _9XNNNNNN,1,Dial(Zap/g2/${EXTEN:1},20,t)
exten => _9XNNNNNN,2,Playback(invalida)
exten => _9XNNNNNN,3,Hangup

;Salida a teléfono celular

exten => _909NNNNNNN,1,Dial(Zap/g2/${EXTEN:1},20,t)
exten => _909NNNNNNN,2,Playback(invalida)
exten => _909NNNNNNN,3,Hangup
```

Al direccionar el canal de salida como `Zap/g2`, se toma la primera línea disponible correspondiente al grupo 2 (que está formado por todos los módulos FXO).

7.4.7. Funcionalidades adicionales de PBX

Funcionalidades clásicas de PBX

Mediante el archivo `features.conf` se definieron las siguientes funcionalidades clásicas de PBX, actualmente implementadas en el I.I.E.:

- Para **aparcar una llamada**, se debe discar el número 255. Luego de esto, la PBX informa en qué extensión la aparcó. Las posibles extensiones donde aparcar llamadas van de la 251 a la 254. El usuario que quiera levantar la llamada aparcada, deberá discar el número que fue anunciado por la PBX.
- Para **levantar una llamada entrante en un teléfono distinto al que está sonando**, se debe discar el número 500. Para que sea posible levantar la llamada, el interno que suena debe pertenecer al mismo grupo que el interno desde donde se levanta la llamada (el grupo es especificado por medio de la variable `pickupgroup` definida para cada interno en el archivo `sip.conf`).
- Para efectuar una **transferencia directa**, se debe marcar la secuencia `#1`.
- Para efectuar una **transferencia asistida**, se debe marcar la secuencia `#2`.

Por detalles del archivo de configuración, referirse al Apéndice D.

Conferencias multi-usuario

El sistema diseñado incluye la funcionalidad de **conferencias multi-usuario**; esto se definió en el archivo `meetme.conf`.

De acuerdo a nuestra configuración, las conferencias pueden ser accedidas mediante la extensión 1234. Esto se logra en el *dialplan* al definir la aplicación `MeetMe` como primera prioridad para dicha extensión. Esto se ilustra a continuación.

```
;Extensión para conferencias

exten => 1234,1,MeetMe(1)
```

Los números de conferencia y sus respectivas claves de acceso se definen en el archivo `meetme.conf`. En principio para el I.I.E. se definió un único número de conferencia, el 1, con clave de acceso 1111. A continuación se muestra el archivo `meetme.conf` que ilustra esto.

```
;
; Este es el archivo de configuración meetme.conf para
; el servidor asterisk.fing.edu.uy
;
; Este archivo de configuración es accedido cada vez que se
; invoca a la aplicación meetme()
;

[rooms]
;
; Formato: conf => número_para_conferencias[,clave]
; [,clave_de_administrador]
;
conf => 1,1111
```

Llamado general (all-call)

En el sistema actualmente implementado, existe la funcionalidad de llamado general (*all-call*), que se activa al marcar la secuencia #5.

Cuando una persona activa el *all-call*, lo que dice por el micrófono de su teléfono se reproduce en los parlantes de todos los teléfonos conectados a la central telefónica del I.I.E..

Para implementar dicha funcionalidad, son necesarios terminales con la capacidad de contestación automática (*auto answer*) y parlantes para reproducir el audio. Dado que en la solución

propuesta se ha optado por utilizar terminales sencillos conectados a los Sipura, este requerimiento no puede cumplirse.

Una forma alternativa encontrada para hacerlo consiste en utilizar un CPA-7B de Viking [29], intercomunicador que se conecta a una interfaz FXS. Para esta conexión pueden utilizarse módulos Zaptel o bien ATAs SIP conectados directamente a la red Ethernet.

Este dispositivo posee dicha capacidad de auto contestación y reproduce el audio por los diferentes parlantes que se distribuyan en el I.I.E. conectados a él.

Portero eléctrico

Para brindar la funcionalidad de apertura de la puerta del I.I.E. a través del sistema basado en *, pueden utilizarse dispositivos como el V-2901A de Valcom [30].

Este dispositivo se conecta a la PBX a través de una interfaz FXO de *. En la arquitectura planteada, se tienen 5 líneas urbanas y 6 módulos FXO; por lo tanto, habría un módulo libre que puede aprovecharse para este fin. Luego, el portero eléctrico se configura en * como una extensión más.

El V-2901A permite destrancar la puerta por medio de una salida de relé que activa la cerradura eléctrica de la misma. El relé es energizado cuando el dispositivo recibe una determinada secuencia de DTMFs (configurable, de máximo 7 tonos).

A nivel del *dialplan*, esto se implementaría por medio de las siguientes líneas en `extensions.conf`, suponiendo que el canal FXO 8 es el usado con este objetivo:

```
exten => *6,1,Dial(Zap/8/${EXTEN},20)
exten => *6,2,Hangup
```

Por otro lado, para que las personas puedan ser atendidas a través del intercomunicador cuando tocan el timbre, se puede definir un contexto `[portero]` que maneje las llamadas provenientes del portero eléctrico. Tal como está implementado hoy en el I.I.E., mediante el *dialplan* para este contexto se puede definir que suene un conjunto predefinido de internos (secretaría, laboratorio de software, etc.).

Para tener esta funcionalidad, es necesario agregarle un intercomunicador al sistema. Uno que interactúa correctamente con el V-2901A es el V-1072A.

8. CONCLUSIONES

En este último capítulo se hace un balance final del proyecto SIP & *, analizando el cumplimiento de los objetivos planteados. También se describen algunos puntos más allá del alcance de este proyecto, pero que son de interés para continuar con las investigaciones en el área, ya sea dentro del I.I.E. o motivando nuevos proyectos de grado.

Finalmente se analizan algunos aspectos relativos a la gestión del proyecto: comparación entre la ejecución real y lo planificado en un principio, la metodología de seguimiento con los tutores y otros temas afines.

8.1. El balance final

El balance global del proyecto SIP & * ha sido muy positivo, ya que se han alcanzado los objetivos planteados dentro del plazo máximo establecido para su ejecución. El gran objetivo del proyecto, era poder llevar a cabo implementaciones fluidas de VoIP en base al protocolo SIP utilizando * como PBX. El enfoque se restringió a redes LAN, con configuraciones de pequeño a mediano porte, ya que no se justificaba atacar grandes escenarios en un primer estudio y por la probable dificultad para conseguir el equipamiento necesario para ello.

En cuanto al protocolo SIP, se concluyó un detallado estudio de sus especificaciones y funcionalidades, pudiéndose corroborar gran parte de sus ventajas en la práctica. También se analizaron, atacaron y resolvieron algunos puntos que comúnmente se asocian a debilidades del protocolo, como es el funcionamiento en escenarios NAT. Para ello se desplegaron diversas topologías de prueba donde se pudo concluir que *, bajo ciertas hipótesis bastante generales, brinda suficientes mecanismos para lograr que las llamadas SIP sean posibles. También ligado a las pruebas de comunicación SIP, se han podido evaluar diferentes UAs, ya sea hardware como el *Cisco ATA 186* o el *Cisco 7905* o directamente *softphones* gratuitos disponibles en Internet.

Por otro lado, se logró la comprensión de la arquitectura de * mediante un estudio que puede separarse en dos niveles:

- en un nivel superior, digamos de usuario, se analizaron los tres elementos constitutivos básicos del sistema:
    - los diferentes canales y sus prestaciones.

    - el *dialplan* con su importancia central como punto de interconexión de los diferentes elementos.
    - el CLI, de gran utilidad para tareas administrativas.

- en más bajo nivel, se buscó el acercamiento a la arquitectura del sistema mediante el estudio del código fuente en el análisis de llamadas SIP. En este punto se trataron:

    - las principales estructuras de datos manejadas.
    - los puntos de interacción entre el núcleo de la PBX y el módulo que implementa el canal SIP.
    - el proceso de inicialización del módulo y la carga de datos del archivo de configuración.
    - el desarrollo de aplicaciones con el objetivo de extender la funcionalidad de *.

Esto era un punto de partida básico para desarrollar extensiones al sistema y detectar problemas de funcionamiento. Además esto permitirá el futuro abordaje de otros canales con gran parte de trabajo ya digerido.

Para aplicar todo el conocimiento adquirido sobre el sistema, se buscó llevar a cabo un proceso completo de ingeniería a través del diseño de un sistema basado en *, con el objetivo de cumplir con los requerimientos reales de telefonía del I.I.E.. Esto condujo al proceso de elección de hardware adecuado (analizando prestaciones y costos) y de configuración de la PBX mediante funcionalidad avanzada del *dialplan*. En este punto, dimos indirectamente con una de las principales ventajas de *: la posibilidad de desarrollar aplicaciones en C a la medida de las necesidades requeridas por el sistema. Por ejemplo, para la implementación puntual de la redirección de internos (no soportada directamente por *), se desarrolló una aplicación acorde a las necesidades que cargada como módulo en el sistema brindaba la funcionalidad faltante. Lamentablemente este diseño no se puso en práctica más que en un escenario de pruebas y a menor escala, para validar todo el funcionamiento de lo configurado. Sin embargo, no se descarta que a través de algún convenio futuro surja la posibilidad de implementar un diseño similar que sea llevado completamente a la práctica.

También se implementaron los cambios en el código para dar soporte IPv6 al canal SIP de *, tomando como punto de partida un *patch* existente. Esto se hizo tanto para la versión estable actual (Asterisk Release 1.0.9) como para la última versión en desarrollo (CVS HEAD). Según se nos ha informado, hay gran interés por esta funcionalidad en diversas universidades en Estados Unido y Asia. Luego de compartir estas modificaciones en la lista de desarrollo de *, se nos recomendó formalizar el aporte enviándolo al *Bugtracker*, donde es evaluado como nueva funcionalidad por los expertos en el desarrollo de *.

A la fecha de completar esta documentación, el *patch* ha sido enviado, pero aún se encuentra pendiente de revisión ya que en general se priorizan aquellos aportes que resuelven problemas graves antes que las nuevas funcionalidades. Sería muy meritorio que este aporte sea

incluido en la versión oficial de distribución de *.

Finalmente, es importante mencionar que se ha adquirido una gran experiencia, nueva para el I.I.E., en cuanto a trabajar en base a un gran proyecto Open Source. Esto presenta varias ventajas, entre ellas la disponibilidad de información, y la posibilidad de realizar consultas a una gran comunidad de usuarios que muy probablemente ya se hayan topado con los problemas que se presentan, y que no dudan en responder. Sin embargo, tiene el inconveniente de que muchas veces uno se encuentra trabajando en un aspecto que muy probablemente también está siendo abordado por la comunidad. Debido a su rápido avance, es posible que su estudio termine antes que el propio, lo cual puede resultar frustrante.

8.2. Próximos pasos

Una de las principales motivaciones del proyecto SIP & * era iniciar un camino de investigación dentro del I.I.E. en el área de VoIP, haciendo especial énfasis en la implementación de soluciones y el desarrollo de aplicaciones que contemplen los requerimientos de un mercado en gran expansión. * es una plataforma ideal para cumplir con estos objetivos, sobretodo en situaciones donde las limitaciones presupuestales imponen fuertes restricciones ante la elección de otras opciones.

Algunos grandes puntos que creemos pueden ser interesantes para continuar con el estudio de estos temas son:

- **IAX2 y *.** IAX2 (*Inter Asterisk Exchange*) es el protocolo VoIP propio de *. Pese a no ser un estándar, es un protocolo abierto que tiene grandes ventajas en algunos aspectos sobre SIP y H.323, como puede ser la operación en presencia de NAT/*Firewall* y el uso eficiente del ancho banda pudiendo trabajar en modo troncal con gran cantidad de llamadas multiplexadas. Por lo tanto, el estudio del protocolo, la implementación del canal, y un análisis comparativo de sus ventajas a la hora de interconectar varias PBX en un escenario de mediano a gran porte pueden ser algunos elementos a considerar en una futura investigación.
- **Desarrollo de un discador predictivo para *.** En la etapa final del proyecto SIP & *, a la hora de elegir una aplicación para desarrollar y extender las funcionalidades de * nos encontramos ante dos posibilidades: el desarrollo de un discador predictivo SIP para * implementado completamente en C[1] o el desarrollo de un *patch* para brindar soporte IPv6 en el canal SIP. La primera tenía un fuerte atractivo comercial ya que existen requerimientos reales en el mercado para un módulo de discado predictivo. Ya habría interés de un grupo de estudiantes por iniciar un proyecto de grado teniendo como objetivo el desarrollo de dicho módulo.
- ***** para *call centers*.** * brinda funcionalidad para el manejo de colas de espera, implementando diferentes estrategias para el reparto de carga entre los agentes (término utilizado

[1] Ya existe algo desarrollado con este objetivo, conocido como *Shady Dial*, pero no totalmente en C o usando bases de dato MySQL. Por información adicional consultar [26]

en el contexto de *, para describir a quienes atienden las llamadas procedentes de las colas). Los paquetes para *call centers* están siendo muy populares hoy en día y la arquitectura abierta de * es ideal para continuar desarrollando mejoras en este ámbito. A modo de ejemplo, presentamos un caso particular donde se han hecho mejoras en este aspecto, incrementando el valor agregado de * para este tipo de aplicaciones. Se ha desarrollado una extensión para * que informa a los integrantes de una cola su tiempo estimado de espera antes de ser atendidos. En vez de aguardar todo ese lapso de tiempo, se les da la posibilidad de cortar y llamar de nuevo sin perder su lugar en la cola[2]. Esto trae las ventajas de liberar las líneas con la posibilidad de incorporar más llamadas a las colas y que los usuarios que llaman ahorren tiempo y dinero.

- **SIP Express Router (SER) y *.** SER es un servidor SIP Open Source que actúa como *proxy*, *registrar* o *redirect*. SER parece soportar conexiones SIP con mayores prestaciones y escalabilidad que *. Por lo tanto, un escenario muy utilizado y cuyo estudio puede ser de gran interés, es el de interconectar un SER con * para que grandes cantidades de clientes SIP puedan, por ejemplo, acceder a la PSTN. El estudio del SER no ha sido abordado en este proyecto ya que es una pieza de software de complejidad comparable al *, pero parece razonable manejarlo como una opción, sobretodo para grandes instalaciones basadas en SIP.

8.3. Gestión del proyecto

El proyecto SIP & * comenzó formalmente en el primer semestre del año 2004 con el curso de *Gestión de Proyectos*. Como trabajo final requerido por dicho curso, para Mayo del mismo año se elaboró un plan de proyecto con una estimación primaria de la secuencia de tareas a desarrollar y sus duraciones. En dicha planificación se estimó un período de ejecución comprendido entre Mayo de 2004 y mediados de Abril de 2005, con un requerimiento de 1500 horas hombre para realizar las tareas.

En su transcurso real, el proyecto se extendió hasta mediados de Agosto de 2005, debido principalmente a razones de índole laboral que afectaron a los tres miembros del grupo. Otros aspectos que en principio podían surgir como factores de retraso; dedicación a cursos y/o exámenes pendientes, falta de materiales para pruebas o disponibilidad de fondos para adquirirlos, mala coordinación y asignación de tareas dentro del grupo, no han sido relevantes. Para el último período comprendido desde Febrero a Agosto de 2005, donde la dedicación al proyecto fue notoriamente más intensa, se hizo un seguimiento detallado de las horas hombre dedicadas obteniendo un total de 900. Por lo tanto, 1400 horas hombre sería una razonable estimación para el total dedicado en el transcurso del proyecto. La extensión del período de ejecución, junto a la dedicación real de una cantidad menor de horas hombre que las planificadas, sugieren dos cosas: una planificación no del todo acertada y una utilización más eficiente de los recursos.

[2] Por información adicional consultar [27]

Una vez finalizado el proyecto, con la posibilidad de evaluar en retrospectiva, queda muy clara la complejidad de realizar un plan de tareas a largo plazo que se ajuste a la realidad. Se han tenido que realizar diversos cambios y tomar decisiones sobre la marcha que no podían ser tenidas en cuenta antes de empezar. Un ejemplo claro fue el proceso de elección de la extensión a desarrollar para *. De acuerdo al plan original, se pensaba implementar el soporte de transferencias asistidas basadas en las recomendaciones de la IETF. Llegado el momento de comenzar a desarrollar esta funcionalidad, se descubrió que la misma ya había sido resuelta por la comunidad, por lo que debió buscarse una alternativa.

En este punto se consideraron las opciones de desarrollar un discador predictivo o el soporte IPv6 para el canal SIP. Este último resultó una opción interesante debido al descubrimiento de un *bounty* publicado por un usuario solicitando esta funcionalidad. Esto no podía preverse a priori, y tampoco la existencia de un *patch* que pudiera ser modificado y corregido para adaptarse a las últimas versiones de *. Esto ejemplifica cómo algunos factores externos, muchas veces impensados, pueden afectar la ejecución de un proyecto apartándolo del plan establecido.

En cuanto al seguimiento del avance del proyecto con los tutores, en Agosto de 2004 se realizó una presentación que cerraba la primera gran etapa del proyecto. Se expusieron los elementos estudiados de SIP como protocolo VoIP y las primeras pruebas de comunicación realizadas con *, en lo que refiere a llamadas SIP.

Finalizada esta etapa se dio un período de aproximadamente 3 meses de muy baja dedicación al proyecto, motivado por razones principalmente de índole laboral. En el período complementario, no se formalizaron más entregables, sino que fue haciendo el seguimiento a través de reuniones e intercambios periódicos para evaluar los resultados y tomar decisiones sobre los siguientes pasos a dar.

Para el proyecto SIP & * no compete un análisis de costos. El equipamiento necesario para realizar pruebas fue todo gentilmente prestado: las tarjetas Digium fueron cedidas desinteresadamente por Teledata S.A.; Softnet Logical Uruguay S.A. nos brindó el teléfono SIP y el Cisco ATA 186; las PCs, los teléfonos y la red de pruebas fueron un aporte en conjunto del grupo de proyecto y el I.I.E..

APÉNDICES

A. ESTRUCTURA BÁSICA PARA APLICACIONES EN *

En la presente sección se incluye el código que se distribuye como punto de partida para el desarrollo de aplicaciones para *. En la Sección 3.4 se hace referencia a este fragmento de código.

```
/*
 * Asterisk -- A telephony toolkit for Linux.
 *
 * Skeleton application
 *
 * Copyright (C) <Year>, <Your Name Here>
 *
 * <Your Name Here> <<You Email Here>>
 *
 * This program is free software, distributed under the terms of
 * the GNU General Public License
 */

#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#include "asterisk.h"

ASTERISK_FILE_VERSION(__FILE__, "$Revision: 1.12 $")

#include "asterisk/file.h"
#include "asterisk/logger.h"
#include "asterisk/channel.h"
#include "asterisk/pbx.h"
#include "asterisk/module.h"
#include "asterisk/lock.h"
#include "asterisk/app.h"

static char *tdesc = "Trivial skeleton Application";
static char *app = "Skel";
static char *synopsis =
"Skeleton application.";
static char *descrip = "This application is a template to build other
applications from.\n"
 " It shows you the basic structure to create your own Asterisk
 applications.\n";

#define OPTION_A        (1 << 0)        /* Option A */
```

```
#define OPTION_B        (1 << 1)        /* Option B(n) */
#define OPTION_C        (1 << 2)        /* Option C(str) */
#define OPTION_NULL        (1 << 3)        /* Dummy Termination */

AST_DECLARE_OPTIONS(app_opts,{
        ['a'] = { OPTION_A },
        ['b'] = { OPTION_B, 1 },
        ['c'] = { OPTION_C, 2 }
});

STANDARD_LOCAL_USER;

LOCAL_USER_DECL;

static int app_exec(struct ast_channel *chan, void *data)
{
        int res = 0;
        struct ast_flags flags;
        struct localuser *u;
        char *options=NULL;
        char *dummy = NULL;
        char *args;
        int argc = 0;
        char *opts[2];
        char *argv[2];

        if (!(args = ast_strdupa((char *)data))) {
                ast_log(LOG_ERROR, "Out of memory!\n");
                return -1;
        }

        if (!data) {
                ast_log(LOG_WARNING, "%s requires an argument (dummy|[options])
                        \n",app);
                return -1;
        }

        LOCAL_USER_ADD(u);
        if ((argc = ast_separate_app_args(args, '|', argv, sizeof(argv) /
                                          sizeof(argv[0])))) {
                dummy = argv[0];
                options = argv[1];
                ast_parseoptions(app_opts, &flags, opts, options);
        }

        if (dummy && !ast_strlen_zero(dummy))
                ast_log(LOG_NOTICE, "Dummy value is : %s\n", dummy);

        if (ast_test_flag(&flags, OPTION_A))
                ast_log(LOG_NOTICE, "Option A is set\n");

        if (ast_test_flag(&flags, OPTION_B))
                ast_log(LOG_NOTICE,"Option B is set with : %s\n", opts[0] ?
                        opts[0] : "<unspecified>");
```

```
        if (ast_test_flag(&flags, OPTION_C))
                ast_log(LOG_NOTICE,"Option C is set with : %s\n", opts[1] ?
                        opts[1] : "<unspecified>");

        /* Do our thing here */
        LOCAL_USER_REMOVE(u);
        return res;
}

int unload_module(void)
{
        STANDARD_HANGUP_LOCALUSERS;
        return ast_unregister_application(app);
}

int load_module(void)
{
        return ast_register_application(app, app_exec, synopsis, descrip);
}

char *description(void)
{
        return tdesc;
}

int usecount(void)
{
        int res;
        STANDARD_USECOUNT(res);
        return res;
}

char *key()
{
        return ASTERISK_GPL_KEY;
}
```

B. CÓDIGO DE LA APLICACIÓN PRINTTEXT

```
/*
 * Asterisk -- A telephony toolkit for Linux.
 *
 * PrintText application
 *
 * Copyright (c) 2004 Grupo SIP & *.  All rights reserved.
 *
 * $Id: app_printtext.c,v 1.1 2004/12/30 18:52:14 citats Exp $
 *
 * This code is released by the author with no restrictions on usage.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <asterisk/file.h>
#include <asterisk/logger.h>
#include <asterisk/options.h>
#include <asterisk/channel.h>
#include <asterisk/pbx.h>
#include <asterisk/utils.h>
#include <asterisk/callerid.h>
#include <asterisk/module.h>
#include <asterisk/config.h>
#include <time.h>


/* Maximum length of any variable */
#define MAXRESULT        1024

static char *tdesc = "Print Text";

static char *app = "PrintText";

static char *synopsis = "PrintText()";

static char *descrip =
"PrintText\n"
"  Print a text previously configured in the\n"
"configuration file as an Asterisk WARNING\n";

#define PRTEXT_CONFIG "printtext.conf"
```

```
STANDARD_LOCAL_USER;

LOCAL_USER_DECL;

static char lang[80] = "en";
static char deftexten[80] = "HelloWorld";
static char deftextes[80] = "HolaMundo";
static char text[80]="";

AST_MUTEX_DEFINE_STATIC(lock);

static int printtext_exec(struct ast_channel *chan, void *data)
{
ast_log(LOG_WARNING, "El texto a desplegar es:\n %s\n", text);
return 0;
}


static int load_config(void)
{
  struct ast_config *cfg;
  struct ast_variable *var;
  char *s;

  ast_mutex_lock(&lock);

  cfg = ast_load(PRTEXT_CONFIG);
  if (!cfg) {
        ast_log(LOG_WARNING, "No se pudo cargar la conf para la
                aplicación PrintText: %s\n", PRTEXT_CONFIG);
        strncpy(text, deftexten, sizeof(text) - 1);
        return -1;
  };

  var = ast_variable_browse(cfg, "general");
  if (!var) {
        /* nada configurado */
        ast_log(LOG_WARNING,"No existe general.\n");
        strncpy(text, deftexten, sizeof(text) - 1);
        return -1;
  }

  if (!(s=ast_variable_retrieve(cfg, "general", "lang"))) {
        ast_log(LOG_WARNING,"Lenguaje no especificado.
                Tomo lenguaje por defecto.\n");
         } else {
        ast_log(LOG_WARNING,"Lenguaje especificado.\n");
        strncpy(lang, s, sizeof(lang) - 1);
  }

  if (!(s=ast_variable_retrieve(cfg, "general", "text"))) {
        ast_log(LOG_WARNING,"Texto no especificado.
                Tomo texto por defecto.\n");
```

```
        if (!strcmp(lang,"es"))
        {
        ast_log(LOG_WARNING,"Lang=%s\n",lang);
        strncpy(text, deftextes, sizeof(text) - 1);
        }else{
        ast_log(LOG_WARNING,"Lenguaje por defecto:%s....\n",lang);
        strncpy(text, deftexten, sizeof(text) - 1);
        }
  } else {
        ast_log(LOG_WARNING,"Texto especificado.\n");
        strncpy(text, s, sizeof(text) - 1);
  }

  ast_mutex_unlock(&lock);
  return -1;

}

int load_module(void)
{
  int res;
  if (!load_config()) {
        return 0;
  }
  res = ast_register_application(app, printtext_exec, synopsis, descrip);
  return res;
}

int reload(void)
{
        load_config();
        return 0;
}

int unload_module(void)
{
        STANDARD_HANGUP_LOCALUSERS;
        return ast_unregister_application(app);
}

char *description(void)
{
  return tdesc;
}

int usecount(void)
{
  int res;
  STANDARD_USECOUNT(res);
  return res;
}

char *key()
{
```

```
  return ASTERISK_GPL_KEY;
}
```

C. ESTRUCTURAS DE DATOS PARA EL CANAL SIP DE *

En esta sección se incluyen las definiciones de las estructuras de datos relevantes para el funcionamiento del canal SIP de *. Son simplemente una referencia, que puede ser de utilidad para la lectura de 4.3.

En primer lugar consideremos las estructuras `sip_user` y `sip_peer` que almacenan los datos ingresados para cada *user* y *peer* en el archivo de configuración `sip.conf`. Se almacenan en estructuras del tipo *linked list*, a saber, `userl` y `peerl` respectivamente. Los datos de estas estructuras se actualizan al inicializar *, o realizar una recarga de la configuración desde el *CLI*.

```
/* Structure for SIP user data. User's place calls to us */
struct sip_user {
    /* Users who can access various contexts */
    char name[80];
    char secret[80];
    char md5secret[80];
    char context[AST_MAX_EXTENSION];
    char callerid[80];
    char accountcode[20];
    char language[MAX_LANGUAGE];
    char musicclass[MAX_LANGUAGE];  /* Music on Hold class */
    char useragent[256];        /* User agent in SIP request */
    unsigned int callgroup;
    unsigned int pickupgroup;
    struct ast_codec_pref prefs;    /* codec prefs */
    int nat;
    int hascallerid;
    int amaflags;
    int insecure;
    int canreinvite;
    int capability;
    int dtmfmode;
    int inUse;
    int incominglimit;
    int outUse;
    int outgoinglimit;
    int promiscredir;
    int restrictcid;
    int trustrpid;
    int progressinband;
    struct ast_ha *ha;
```

```
    struct sip_user *next;
};
```

```
/* Structure for SIP peer data, we place calls to peers if
registred  or fixed IP address (host) */
struct sip_peer {
    char name[80];
    char secret[80];
    char md5secret[80];
    char context[AST_MAX_EXTENSION];    /* JK02: peers need context too to
                                        allow parking etc */
    char username[80];
    char tohost[MAXHOSTNAMELEN];
    char regexten[AST_MAX_EXTENSION];   /* Extension to register (if
                                        regcontext is used) */
    char fromuser[80];
    char fromdomain[MAXHOSTNAMELEN];
    char fullcontact[256];
    char mailbox[AST_MAX_EXTENSION];
    char language[MAX_LANGUAGE];
    char musicclass[MAX_LANGUAGE];  /* Music on Hold class */
    char useragent[256];        /* User agent in SIP request */
    struct ast_codec_pref prefs;    /* codec prefs */
    int lastmsgssent;
    time_t  lastmsgcheck;
    int dynamic;
    int expire;
    int expiry;
    int capability;
    int rtptimeout;
    int rtpholdtimeout;
    int insecure;
    int nat;
    int canreinvite;
    unsigned int callgroup;
    unsigned int pickupgroup;
    int promiscredir;
    int dtmfmode;
    int trustrpid;
    int progressinband;
    net_sockaddr *addr; /* IP address of peer */
    net_sockaddr *mask; /* */

    /* Qualification */
    struct sip_pvt *call;       /* Call pointer */
    int pokeexpire;             /* When to expire poke */
    int lastms;                 /* How long last response took (in ms),
                                or -1 for no response */
    int maxms;                  /* Max ms we will accept for the host to be
                                up, 0 to not monitor */
    struct timeval ps;          /* Ping send time */
```

```
    net_sockaddr *defaddr;  /* Default IP address, used until registration */
    struct ast_ha *ha;
    int delme;
    int selfdestruct;
    int lastmsg;
    int temponly;
    struct sip_peer *next;
};
```

La estructura `sip_request` guarda toda la información de los mensajes SIP (encabezados, contenido SDP).

```
/* sip_request: The data grabbed from the UDP socket */
struct sip_request {
  char *rlPart1; /* SIP Method Name or "SIP/2.0" protocol version */
  char *rlPart2; /* The Request URI or Response Status */
  int len;
  int headers;                    /* SIP Headers */
  char *header[SIP_MAX_HEADERS];
  int lines;                      /* SDP Content */
  char *line[SIP_MAX_LINES];
  char data[SIP_MAX_PACKET];
};
```

La estructura `sip_pvt` mantiene la información de cada llamada. Almacena datos específicos de la tecnología SIP y dado que cada llamada utiliza un canal, incluye un puntero `owner` del tipo `ast_channel` para vincularse con el canal al cual pertenece. Todas las llamadas existentes en cierto instante se almacenan en una *linked list*, `iflist`.

```
/* sip_pvt: PVT structures are used for each SIP conversation*/
static struct sip_pvt {
    ast_mutex_t lock;           /* Channel private lock */
    char callid[80];            /* Global CallID */
    char randdata[80];          /* Random data */
    struct ast_codec_pref prefs; /* codec prefs */
    unsigned int ocseq;         /* Current outgoing seqno */
    unsigned int icseq;         /* Current incoming seqno */
    unsigned int callgroup;         /* Call group */
    unsigned int pickupgroup;       /* Pickup group */
    int lastinvite;             /* Last Cseq of invite */
    int alreadygone;            /* Whether or not we've already
                                been destroyed by or peer */
    int needdestroy;            /* if we need to be destroyed */
    int capability;             /* Special capability (codec) */
    int novideo;                /* Didn't get video in invite,
                                don't offer */
```

```
int jointcapability;            /* Supported capability at
                                both ends (codecs ) */
int peercapability;             /* Supported peer capability */
int prefcodec;                  /* Preferred codec (outbound only) */
int noncodeccapability;
int outgoing;                   /* Outgoing or incoming call? */
int authtries;                  /* Times we've tried to authenticate */
int insecure;                   /* Don't check source port/ip */
int expiry;                /* How long we take to expire */
int branch;                /* One random number */
int canreinvite;                /* Do we support reinvite */
int ringing;                    /* Have sent 180 ringing */
int progress;                   /* Have sent 183 message progress */
int tag;                   /* Another random number */
int nat;                   /* Whether to try to support NAT */
int sessionid;                  /* SDP Session ID */
int sessionversion;             /* SDP Session Version */
net_sockaddr *sa;          /* Our peer */
net_sockaddr *redirip;          /* Where our RTP should be going if
                                not to us */
net_sockaddr *vredirip;         /* Where our Video RTP should be going
                                if not to us */
int redircodecs;                    /* Redirect codecs */
net_sockaddr *recv;        /* Received as */
net_sockaddr *ourip;            /* Our IP */
struct ast_channel *owner;          /* Who owns us */
char exten[AST_MAX_EXTENSION];      /* Extension where to start */
char refer_to[AST_MAX_EXTENSION];   /* Place to store REFER-TO
                                    extension */
char referred_by[AST_MAX_EXTENSION];    /* Place to store REFERRED-BY
                                        extension */
char refer_contact[AST_MAX_EXTENSION];  /* Place to store Contact info
                                        from a REFER extension */
struct sip_pvt *refer_call;     /* Call we are referring */
struct sip_route *route;        /* Head of linked list of routing steps
                                (fm Record-Route) */
int route_persistant;           /* Is this the "real" route? */
char from[256];                 /* The From: header */
char useragent[256];                /* User agent in SIP request */
char context[AST_MAX_EXTENSION];    /* Context for this call */
char fromdomain[MAXHOSTNAMELEN];    /* Domain to show in the from
                                    field */
char fromuser[AST_MAX_EXTENSION];   /* User to show in the user field */
char fromname[AST_MAX_EXTENSION];   /* Name to show in the user field */
char tohost[MAXHOSTNAMELEN];        /* Host we should put in the
                                    "to" field */
char language[MAX_LANGUAGE];        /* Default language for this call */
char musicclass[MAX_LANGUAGE];          /* Music on Hold class */
char rdnis[256];                /* Referring DNIS */
char theirtag[256];             /* Their tag */
char username[256];
char peername[256];
char authname[256];             /* Who we use for authentication */
char uri[256];                  /* Original requested URI */
```

```
    char peersecret[256];
    char peermd5secret[256];
    char callerid[256];         /* Caller*ID */
    int restrictcid;            /* hide presentation from remote user */
    char via[256];
    char fullcontact[128];          /* Extra parameters to go in the "To"
                                    header */
    char accountcode[20];           /* Account code */
    char our_contact[256];          /* Our contact header */
    char realm[MAXHOSTNAMELEN];     /* Authorization realm */
    char nonce[256];            /* Authorization nonce */
    char opaque[256];           /* Opaque nonsense */
    char qop[80];               /* Quality of Protection, since SIP wasn't
                                complicated enough yet. */
    char domain[MAXHOSTNAMELEN];        /* Authorization domain */
    char lastmsg[256];          /* Last Message sent/received */
    int amaflags;               /* AMA Flags */
    int pendinginvite;          /* Any pending invite */
    int needreinvite;           /* Do we need to send another reinvite? */
    int pendingbye;             /* Need to send bye after we ack? */
    int gotrefer;               /* Got a refer? */

    struct sip_request initreq;     /* Initial request */

    int maxtime;                /* Max time for first response */
    int initid;             /* Auto-congest ID if appropriate */
    int autokillid;             /* Auto-kill ID */
    time_t lastrtprx;           /* Last RTP received */
    int rtptimeout;             /* RTP timeout time */
    int rtpholdtimeout;         /* RTP timeout when on hold */

    int subscribed;
        int stateid;
    int dialogver;
    int promiscredir;           /* Promiscuous redirection */

    int trustrpid;
    int progressinband;

    int dtmfmode;
    struct ast_dsp *vad;

    struct sip_peer *peerpoke;      /* If this calls is to poke a peer,
                                    which one */
    struct sip_registry *registry;      /* If this is a REGISTER call, to
                                        which registry */
    struct ast_rtp *rtp;            /* RTP Session */
    struct ast_rtp *vrtp;           /* Video RTP session */
    struct sip_pkt *packets;        /* Packets scheduled for
                                    re-transmission */
    struct sip_history *history;    /* History of this SIP dialog */
    struct sip_pvt *next;           /* Next call in chain */
} *iflist = NULL;
```

D. ARCHIVOS DE CONFIGURACIÓN DEL SISTEMA DESARROLLADO PARA EL IIE

A continuación se presentan los archivos de configuración del sistema diseñado para sustituir la central telefónica existente en el IIE. Los mismos son `sip.conf`, `zapata.conf`, `voicemail.conf`, `extensions.conf` y `features.conf`.

En el caso de `sip.conf` y `voicemail.conf`, sólo se muestran las configuraciones de la extensión 600 a modo de ejemplo, en el entendido de que la configuración del resto de las extensiones es idéntica. Análogamente, para el caso de `zapata.conf`, sólo se muestran las líneas que agregamos nosotros para la configuración de las interfaces que tendremos.

```
;
; Este es el archivo de configuración sip.conf para
; el servidor asterisk.fing.edu.uy
;

[general]

;Puerto donde escuchar por mensajes SIP
port=5060

;Dirección IP donde escuchar por mensajes SIP
bindaddr=0.0.0.0    ;Todas las direcciones de la máquina

;Deshabilitar todos los codecs
disallow=all

;Habilitar el codec G.729
allow=g729

;Contexto a donde enviar las llamadas SIP no identificadas
context=bogon-calls


[600]

;Este usuario puede realizar y recibir llamadas
type=friend

;Nombre de usuario
username=600
```

```
;Clave de usuario
secret=1111

;Este host puede utilizar diferentes IPs
host=dynamic

;Contexto a donde enviar sus llamadas
context=desde-sip

;Indicación de mensajes de voz
mailbox=600

;Grupo que identifica que llamadas puede levantar
pickupgroup=1

;Grupo que identifica sus llamadas para poder ser levantadas
callgroup=1

;Modo para el manejo de DTMF
dtmfmode=rfc2833
```

```
;
; Este es el archivo de configuración zapata.conf para
; el servidor asterisk.fing.edu.uy
;

;
;Módulos FXS
;

;Tipo de señalización (FXO para módulos FXS)
signalling=fxo_ks

;Habilitar cancelación de eco
echocancel=yes
echocancelwhenbridged=yes
echotraining=400

;No modificar el callerid
callerid=asreceived

;Grupo al cual pertenecen estos módulos
group=1

;Contexto a donde enviar sus llamadas
context=emergencia

;Ubicación del módulo FXS en la primera placa
channel=>1

;Ubicación del módulo FXS en la segunda placa
channel=>5
```

```
;
;Módulos FXO
;

;Tipo de señalización (FXS para módulos FXO)
signalling=fxs_ks

;Grupo al cual pertenecen estos módulos
group=2

;Contexto a donde enviar sus llamadas
context=menu-principal

;Ubicación de módulos FXO en la primera placa
channel=>2-4

;Ubicación de módulos FXO en la segunda placa
channel=>6-8
```

```
;
; Este es el archivo de configuración voicemail.conf para
; el servidor asterisk.fing.edu.uy
;

;
; Configuracion del Correo de Voz
;

[general]

; Formato en que se guardan los mensajes
format=wav

; Campo De: en el mail de notificación de nuevo mensaje
serveremail=voicemail@asterisk.iie.fing.edu.uy

; Enviar el mensaje adjunto en el mail
attach=yes

; Maxima duracion de un mensaje de correo de voz
maxmessage=180

[local]

;Internos de Oficina
;Formato: numero de casilla => password,nombre
;

115 => 1111,115

;Internos personales
;Formato: numero de casilla => password,nombre,mail para enviar
```

```
;mensaje adjunto
;

661 => 1111,Federico Morales,fmorales@fing.edu.uy
```

```
;
; Este es el archivo de configuración extensions.conf para
; el servidor asterisk.fing.edu.uy
;

[general]

;Estas dos líneas deshabilitan la posibilidad de sobreescribir
;esta configuración desde el CLI
static=yes
writeprotect=yes

[bogon-calls]

;Contiene todas las llamadas que vienen de un
;contexto no identificado.
;A todas las llamadas se les dara tono de Congestion.

exten => _.,1,Congestion


[menu-principal]

;Menú de bienvenida

exten =>s,1,ResponseTimeout(5)
exten =>s,2,Background(bienvenida)
exten =>s,3,Background(instrucciones)

exten =>t,1,Background(instrucciones)
exten =>t,2,Background(silence/5)
exten =>t,3,Playback(despedida)
exten =>t,4,Hangup

exten =>i,1,Playback(ext_incorrecta)
exten =>i,2,Goto(menu-principal,t,1)

exten =>9,1,Goto(directorio,s,1)
exten =>*,1,Goto(desde-sip,2999,1)
exten =>_[167]XX,1,Goto(desde-sip,${EXTEN},1)

[directorio]

;Directorio con anuncio de internos de docentes

exten =>s,1,ResponseTimeout(5)
exten =>s,2,DigitTimeout(1)
exten =>s,3,Background(directorio)
```

```
exten =>1,1,DigitTimeout(5)
exten =>1,2,Background(AB/directorio)

exten =>2,1,DigitTimeout(5)
exten =>2,2,Background(C/directorio)

exten =>3,1,DigitTimeout(5)
exten =>3,2,Background(DE/directorio)

exten =>4,1,DigitTimeout(5)
exten =>4,2,Background(FG/directorio)

exten =>5,1,DigitTimeout(5)
exten =>5,2,Background(HIJKL/directorio)

exten =>6,1,DigitTimeout(5)
exten =>6,2,Background(MNO/directorio)

exten =>7,1,DigitTimeout(5)
exten =>7,2,Background(PQ/directorio)

exten =>8,1,DigitTimeout(5)
exten =>8,2,Background(RS/directorio)

exten =>9,1,DigitTimeout(5)
exten =>9,2,Background(T-Z/directorio)

exten =>_1XX,1,Goto(desde-sip,${EXTEN},1)
exten =>_6XX,1,Goto(desde-sip,${EXTEN},1)
exten =>_7XX,1,Goto(desde-sip,${EXTEN},1)

exten => i,1,Goto(menu-principal,i,1)

exten => t,1,Playback(despedida)
exten => t,2,Hangup

[emergencia]

;Teléfonos de emergencia

include => salientes

exten=>800,1,Dial(Zap/1,20,tr)
exten=>800,2,Goto(102)
exten=>800,102,Hangup

exten=>801,1,Dial(Zap/5,20,tr)
exten=>801,2,Goto(102)
exten=>801,102,Hangup

[desde-sip]

;Internos SIP
```

```
include => salientes
include => llamadas_aparcadas

;Secretaría e Internos de Oficina

exten => _1XX,1,Dial(SIP/${EXTEN},20,Tt)
exten => _1XX,2,Voicemail(u${EXTEN})
exten => _1XX,3,Hangup
exten => _1XX,102,Voicemail(b${EXTEN})
exten => _1XX,102,Hangup


;Extensiones Personales

;Obtengo el interno de oficina asociado al interno personal
;El valor se devuelve en la variable INTERNO
;La búsqueda se hace por la clave redirect/${EXTEN}
;donde ${EXTEN} tiene almacenado el valor del interno personal

exten => _[67]XX,1,DBget(INTERNO=redirect/${EXTEN})

;Llamo al interno personal
;Si da Unavailable, salto a la prioridad 3
;Si da Busy, salto a la prioridad 103

exten => _[67]XX,2,Dial(SIP/${EXTEN},20,Tt)

;Llamo al interno de oficina
;Si da Unavailable, salto a la prioridad 4
;Si da Busy, salto a la prioridad 104

exten => _[67]XX,3,Dial(SIP/${INTERNO},20,Tt)


exten => _[67]XX,4,Voicemail(u${EXTEN})
exten => _[67]XX,5,Hangup
exten => _[67]XX,103,Goto(104)
exten => _[67]XX,104,Voicemail(b${EXTEN})
exten => _[67]XX,105,Hangup

;Correo de voz

exten => 2999,1,VoicemailMain(${CALLERIDNUM})

;Extensión para conferencias

exten => 1234,1,MeetMe(1)

[salientes]

;Mantener el tono de línea luego de discar 9
ignorepat =>9
```

```
;Salida a teléfono fijo

exten => _9XNNNNNN,1,Dial(Zap/g2/${EXTEN:1},20,t)
exten => _9XNNNNNN,2,Playback(invalida)
exten => _9XNNNNNN,3,Hangup

;Salida a teléfono celular

exten => _909NNNNNNN,1,Dial(Zap/g2/${EXTEN:1},20,t)
exten => _909NNNNNNN,2,Playback(invalida)
exten => _909NNNNNNN,3,Hangup
```

```
;
; Este es el archivo de configuración features.conf para
; el servidor asterisk.fing.edu.uy
;

[general]

;Extensión a discar para aparcar llamadas
parkext => 255

;Extensiones en las que son aparcadas las llamadas
parkpos => 251-254      ; "Playa de estacionamiento"

;Contexto al cual enviar las llamadas aparcadas
context => llamadas_aparcadas

;Tiempo máximo (en seg.) que una llamada puede permanecer
;aparcada
parkingtime => 60

;Sonido para indicar que se completó una transferencia
xfersound = beep

;Sonido para indicar que falló una transferencia
xferfailsound = beeperr

;Extensión para levantar las llamadas aparcadas
pickupexten = 500

;Tiempo máximo (en ms) entre dígitos para activar las
;funcionalidades
featuredigittimeout = 500


[featuremap]

;
;Aquí se definen las funcionalidades de features.conf
;

;Transferencia directa o incondicional
```

```
blindxfer => #1

;Transferencia asistida
atxfer => #2
```

E. ARCHIVOS DE CONFIGURACIÓN DEL SERVIDOR DNS IPV6 (BIND)

En esta sección se incluyen los archivos de configuración utilizados para el servidor DNS IPv6. En primer lugar, se tiene el archivo `named.conf` donde se define la configuración general del servidor. La zona configurada para las pruebas es `v6.moreira.org`.

```
/*
 * BIND 8.2.2 boot configuration file
 *
 * Author: Bertrand Buclin
 *
 * Modification History:
 *  16-Sep-97   Buclin
 *      Initial Version
 *      31-Jan-00      Sekiya
 *             Revised
 */

options {
    directory "/var/named";
    listen-on-v6 {any;};
};

//
// localhost
//
zone    "localhost" {
        type master;
        file "localhost";
};


//
// IPv4 zone files
//
zone    "." {
        type hint;
        file "root.cache" ;
};

zone    "1.0.0.127.in-addr.arpa"    {
        type master;
        file "localhost";
};
```

```
//
// IPv6 zone files
// ==========
//
// First, load the zone for the IPv6 loopback address.
//
zone
"1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.ip6.int."
{
        type master;
        file "localhost";
};

//
//  If your IPv6 domain is "v6.moreira.org", you need below zone.
//
zone "v6.moreira.org"   {
        type master;
        file "v6.moreira.org";
};

//
//  Reverse for the zone v6.moreira.org
//
zone "6.0.0.c.ip6.arpa" {
        type master;
        file "c006::";
};
```

El siguiente archivo, `v6.moreira.org`, es donde se define la información de la zona.

```
; File:v6.moreira.org.zone
; IPv6 Casa Moreira Organization.
; IP v6 test network
;
@        IN        SOA        ns.v6.moreira.org.        root.v6.moreira.org. (
                                        100013117
                                        3H        ; refresh
                                        15M        ; retry
                                        1W        ; expiry
                                        1D )        ; minimum
        IN        NS                ns.v6.moreira.org.
        IN        MX        10        mail.v6.moreira.org.
;
;
; Network names
;
v6.moreira                IN        AAAA        c006::

;
; Local hosts
; ------------------
asterisk                IN        AAAA        c006::2
```

```
mail                        IN        CNAME        asterisk
;
moreira                        IN        AAAA        c006::3
;
shaka-zulu               IN        AAAA        c006::1
;
```

En el archivo `c006__` se indica la información para las consultas inversas.

```
; File: c006__
; IPv6 reverse lookup zone for c006::/16
; Para la red de pruebas de SIP & *
; Creado el: 5 de Julio de 2005, por Federico Morales
;
;
@        IN        SOA        ns.v6.moreira.org.        root.v6.moreira.org. (
                                        100013117
                                        3H        ; refresh
                                        15M        ; retry
                                        1W        ; expiry
                                        1D )        ; minimum
               IN        NS                ns.v6.moreira.org.
;
; Set the origin to the pTLA prefix.
;
$ORIGIN        6.0.0.c.ip6.arpa.
0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0
                                IN        PTR        v6.moreira
1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0
                                IN        PTR        shaka-zulu.v6.moreira.org.
2.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0
                                IN        PTR        asterisk.v6.moreira.org.
3.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0
                                IN        PTR        moreira.v6.moreira.org.
```

Finalmente, en el archivo `localhost` se indica la información para resolver el *localhost*.

```
;File: localhost
@        IN        SOA        ns.ipv6domain-tottaro.org.
                                root.ipv6domain-tottaro.org. (
                                        3 ; Serial
                                        3H        ; refresh
                                        15M        ; retry
                                        1W        ; expiry
                                        1D )        ; minimum
;
        IN        NS                localhost.
;
localhost.                                IN        A        127.0.0.1
```

```
1.0.0.127.in-addr.arpa.                 IN        PTR                 localhost.
;
localhost.                                  IN         AAAA        ::1
;
$ORIGIN 0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.ip6.int.
1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0         IN        PTR                 localhost.
```

F. CÓDIGO DE LA APLICACIÓN LOADDB

```
/*
 * Asterisk -- A telephony toolkit for Linux.
 *
 * LoadDB application
 *
 * Copyright (c) 2005 Grupo SIP & *.  All rights reserved.
 *
 * $Id: app_loaddb.c,v 0.1 2005/6/01 18:52:14 citats Exp $
 *
 * This code is released by the author with no restrictions on usage.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <asterisk/file.h>
#include <asterisk/logger.h>
#include <asterisk/options.h>
#include <asterisk/channel.h>
#include <asterisk/pbx.h>
#include <asterisk/utils.h>
#include <asterisk/callerid.h>
#include <asterisk/module.h>
#include <asterisk/config.h>
#include <time.h>
#include <asterisk/astdb.h>
#include <asterisk/lock.h>
#include <asterisk/app.h>


static char *tdesc = "Load Data Base";

static char *app = "LoadDB";

static char *synopsis = "LoadDB()";

static char *descrip =
"LoadDB\n"
"  Load information into the local Database\n"
"from a comma separated values file.\n";

static char *file ="/etc/asterisk/files/BaseDatos.csv";
```

```
STANDARD_LOCAL_USER;

LOCAL_USER_DECL;

static int loaddb_exec(struct ast_channel *chan, void *data)
{
        return 0;
}
int reload(void)
{
        FILE *TextFile;
        char *line,*name,*ext,*office;
        ssize_t c;
        size_t line_size = 200;
        /* size_t line_size = 0; */
        int *code;


        if ((TextFile = fopen(file,"r")) == NULL){
                ast_log(LOG_WARNING,"Unable to open file: %s\n",file);
                return -1;
        }
        else {
           while((c = getline(&line,&line_size,TextFile)) != -1) {
                name = strtok(line,";");
                if (!(name==NULL)){
                        office = strtok(NULL,";");
                        if (!(office==NULL)){
                                ext = strtok(NULL,";\0");
                                if (!(ext==NULL)){
                                      \* Aquí se verificaron todos los campos *\
                                  code = ast_db_put("cidname",ext,name);
                                  code = ast_db_put("redirect",ext,office);
                                }else{
                                  fclose(TextFile);
                                  ast_log(LOG_WARNING,
                                          "Formato Incorrecto de Archivo\n");
                                  return 0;
                                }
                        }else{
                                fclose(TextFile);
                                ast_log(LOG_WARNING,
                                        "Formato Incorrecto de Archivo\n");
                                return 0;
                        }
                }else{
                        fclose(TextFile);
                        ast_log(LOG_WARNING,"Formato Incorrecto de Archivo\n");
                        return 0;
                }
           }
           fclose(TextFile);
        }
```

```
        return 0;
}
int unload_module(void)
{
        STANDARD_HANGUP_LOCALUSERS;
        return ast_unregister_application(app);
}

int load_module(void)
{
  return ast_register_application(app, loaddb_exec, synopsis, descrip);
}

char *description(void)
{
  return tdesc;
}

int usecount(void)
{
  int res;
  STANDARD_USECOUNT(res);
  return res;
}

char *key()
{
  return ASTERISK_GPL_KEY;
}
```

BIBLIOGRAFíA

[1] TERENA. *IP Telephony Cookbook*. ©2004 `http://www.terena.nl/library/IPTELEPHONYCOOKBOOK` [visitada en Junio de 2004].

[2] Scott Keagy. *Integración de Redes de Voz y Datos*. Madrid:Cisco Press, 2001.

[3] J. Rosenberg [y otros]. *SIP: Session Initiation Protocol*, RFC 3261. Junio de 2002.[1]

[4] H. Schulzrinne ; J. Rosenberg. *A Comparison of SIP and H.323 for Internet Telephony*, Internet Draft.

[5] Eric Osborne ; Ajay Simha. *Traffic Engineering with MPLS*. Cisco Press:1era. Edición, Julio de 2002.

[6] Andrew S. Tanenbaum. *Redes de Computadoras*. Pearson:3era. Edición, 1997.

[7] RADCOM. *Fine-tuning Voice over Packet services*. 1999. `http://www.radcom.com/radcom/technlgy/pdf/FineTuningVoiceOverPackeServices.pdf` [visitada en Mayo de 2004].

[8] S. Deering ; R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*, RFC 2460. Diciembre de 1998.

[9] R. Hinden ; S. Deering. *IP Version 6 Addressing Architecture*, RFC 2373. Julio de 1998.

[10] Nicolás de León ; Ricardo Siri ; Martín Wolff. *Proyecto IPv6*. Proyecto de Grado. Montevideo, Instituto de Ingeniería Eléctrica, Facultad de Ingeniería, Universidad de la República, Febrero de 2003.

[11] R. Sparks ; A. Johnston ; D. Petrie. *Session Initiation Protocol Call Control - Transfer*, Internet Draft. Abril de 2005.

[12] R. Sparks [y otros]. *Session Initiation Protocol Service Examples*, Internet Draft. Febrero de 2005.

[13] G. Camarillo. *IPv6 Transcition in the Session Initiation Protocol (SIP)*, Internet Draft. Febrero de 2005.

[14] J. Mulahusic ; H. Persson. *SIP Issues in Dual-stack Environments*, Internet Draft. Febrero de 2003.

[1] Todas las RFC de la IETF incluídas en la presente bibliografía pueden ser consultadas en `http://www.ietf.org/rfc.html`.

[15] R. Gilligan [y otros]. *Basic Socket Interface Extensions for IPv6*, RFC 3493. Febrero de 2003.

[16] J. Rosenberg [y otros]. *STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)*, RFC 3489. Marzo de 2003.

[17] J. Rosenberg ; R. Mahy ; C. Huitema. *Traversal Using Relay NAT (TURN)*, Internet Draft. Julio de 2005.

[18] H. Schulzrinne ; S. Petrack. *RTP Payload for DTMF Digits, Telephony Tones and Telephony Signals*, RFC 2833. Mayo de 2000.

[19] voip-info.org. *voip-info.org*. ©2002-2005. `http://voip-info.org` [visitada en Julio de 2005].

[20] voip-info.org. *voip-info.org*. ©2002-2005. `http://voip-info.org/tiki-index.php?page+Asterisk` [visitada en Agosto de 2005].

[21] Digium. *Asterisk — The Open Source PBX*. `http://www.asterisk.org` [visitada en Julio de 2005].

[22] Digium. *Digium - The Asterisk Telephony Company*. `http://www.digium.com` [visitada en Junio de 2005].

[23] Asterisk Documentation Project. *News*. `http://www.asteriskdocs.org` [visitada en Agosto de 2004].

[24] Digium. *The Asterisk-Dev Archives*. `http://lists.digium.com/pipermail/asterisk-dev/` [visitada en Julio de 2005].

[25] Digium. *The Asterisk-Users Archives*.

`http://lists.digium.com/pipermail/ asterisk-users/` [visitada en Julio de 2005].

[26] John Todd. *Shady Dial*. `http://www.loligo.com/asterisk/misc/mirrors/www.dynx.net/ASTERISK/dialer/dialer.txt` [visitada en Mayo de 2005].

[27] voip-info.org. *OrderlyQ*. ©2002-2005. `http://voip-info.org/tiki-index.php?page=OrderlyQ` [visitada en Agosto de 2005].

[28] Sipura Technology Inc. *SPA 2100 Data Sheet*. ©2003-2004. `http://www.sipura.com/Documents/SPA-2100.pdf` [visitada en Octubre de 2004].

[29] Viking. *CPA-7b Technical Practice*. Abril de 2004. `http://www.vikingelectronics.com/products/pdf/cpa-7b.pdf`. [visitada en Agosto de 2005].

[30] Valcom. *V-2901A Universal Door Answering System*. `http://www.valcom.com/pdf/v-2901a.pdf` [visitada en Agosto de 2005].

[31] Asterisk Development Proxy. *Proxy*. 2005. `http://dev.asteriskdocs.org/index.php/Patches` [visitada en Junio de 2005].

[32] Digium. *Mantis* [Sistema Bugtracker de Asterisk]. ©2005. `http://bugs.digium.com` [visitada en Julio de 2005].

[33] voip-info.org. *AstWind*. ©2002-2005. `http://www.voip-info.org/tiki-index.php?page=AstWind` [visitada en Agosto de 2005].

[34] Athoris Srl. *Asterisk - The Open Source PBX - Windows Version* `http://www.asteriskwin32.com` [visitada en Agosto de 2005].