Títol: Stack RTP para dispositivos móviles Symbian

Autor: Daniel Rodríguez Roldán

Director: Juan López Rubio

Data: 24 de febrero de 2006

Resum

El objetivo principal de este proyecto es estudiar las posibilidades de implementar una pila RTP para el sistema operativo de dispositivos móviles Symbian OS.

Durante la realización del proyecto se da una visión teórica del protocolo RTP. Este protocolo se ha convertido en el estándar para la transmisión de contenidos en tiempo real sobre redes IP. Por este motivo se perfila como una posibilidad interesante el adaptar sus funcionalidades a los dispositivos de bolsillo.

Para poder soportar comunicaciones multimedia en tiempo real, es necesaria la existencia de un sistema operativo robusto y eficiente, que pueda sacar el máximo partido de las posibilidades de los terminales. Symbian OS cumple con todos los requisitos. En este proyecto se repasan las cualidades más destacables de este sistema operativo.

Por último, se ha implementado una sencilla aplicación de prueba para medir las posibilidades reales que ofrece la migración.

Todos los módulos que componen el proyecto han sido programados con C++, por lo que ofrecen un diseño orientado a objetos que ha sido descrito mediante el lenguaje gráfico UML.

Title: RTP stack for Symbian mobile devices

Author: Daniel Rodríguez Roldán

Director: Juan López Rubio

Date: February, 24th 2006

Overview

The primary target of this project is to study the possibilities of implementing a RTP stack for the operating system of mobile devices Symbian OS.

During the accomplishment of the project, a theoretical vision of protocol RTP occurs. This protocol has become the standard for the transmission of contents in real time on IP networks. For this reason adapting is outlined like an interesting possibility to adapt their functionalities to the pocket devices.

In order to be able to support the multimedia communications in real time, the existence of a robust and efficient operating system is necessary, that can take maximum advantage of the possibilities of the terminals. Symbian OS fulfills all the requirements. In this project the most remarkable qualities of this operating system are reviewed.

Finally, a simple test application has been implemented to measure the real possibilities that it offers the migration. All the modules that compose the project have been programmed with C++, reason why they offer an Object Oriented design that has been described by means of graphical language UML.

ÍNDICE

CAPÍTULO 1. Introducción

1.1. Oportunidades del proyecto

Existe una tendencia clara en el ámbito de las telecomunicaciones hacia una convergencia de servicios a través de una misma red. Como cabía esperar, el protocolo que pretende englobar los servicios es IP. No extraña esta decisión, ya que IP ha demostrado sobradamente su polivalencia, siendo capaz de adaptarse perfectamente a una red tan anárquica como es Internet. Esta tendencia se conoce como convergencia IP.

La convergencia IP encuentra su máximo exponente en la adaptación de servicios de voz -muy arraigados a las redes de conmutación de circuitos- a una red basada en conmutación de paquetes. Esta tecnología es conocida como VoIP (Voz sobre IP). Su meta es ofrecer servicios de valor añadido a las comunicaciones de voz, así como reducir su coste, pudiendo englobar las comunicaciones de datos, vídeo y voz sobre una misma red.

La aplicación inmediata que surge a raíz de VoIP es la telefonía IP. La posibilidad de tener los servicios clásicos de telefonía sobre una red IP ofrecen ilimitadas ventajas, tales como permitir la movilidad de un mismo usuario entre distintos tipos de terminales, servicios de valor añadido como incluir vídeo en las comunicaciones, etc. Así mismo el servicio de telefonía se aleja de una red inflexible, la cual necesita una gran inversión para modificar sus servicios, a favor de una red que permite a los operadores libertad para ofrecer cualquier tipo de servicio añadido.

Es obvio que un sector en auge, como es el de la telefonía móvil, no puede quedarse a un lado ante esta oportunidad de ampliar sus prestaciones. Para ello se hace necesaria la existencia de una nueva generación de terminales móviles, que dejen a un lado los formatos propietarios y empiecen a soportar cualquier estándar de comunicación. Es en este aspecto donde un sistema operativo para móviles de alta capacidad, capaz de soportar cualquier tipo de estándar, se hace indispensable.

Como podremos ver en este proyecto, el sistema operativo para móviles Symbian OS cumple con los requisitos necesarios para soportar esta convergencia. No obstante carece de una pila del protocolo RTP, que se ha convertido en el protocolo estándar para la transmisión de contenidos en tiempo real a través de redes IP.

Este proyecto se basa en el estudio de la posibilidad de implementar una pila RTP para Symbian OS. Esto constituiría el primer paso para la implantación de aplicaciones en tiempo real, utilizando la conectividad IP que ofrecen los terminales. La proliferación de este tipo de servicios podrían hacer converger la telefonía móvil con la fija de forma completa, permitiendo a un terminal realizar llamadas a través de la red UMTS y, a la vez, dando la posibilidad al terminal de conectarse a la red fija a través de, por ejemplo, WLAN.

1.2. Objetivos

El objetivo principal de este proyecto es estudiar las posibilidades de implementar una pila RTP, para el sistema operativo de dispositivos móviles Symbian OS. Para ello se pretende adaptar una librería RTP ya implementada. Con el fin de alcanzar este objetivo, podemos dividir el proyecto en cinco fases.

Una primera fase, dedicada al estudio teórico del protocolo. Esta fase tiene como objetivo conocer a fondo los aspectos más relevantes de RTP, para poder comprender y solventar los posibles errores que se produzcan durante la implementación de la pila RTP.

A continuación, y siguiendo con la línea de estudio teórico, se pretende estudiar la plataforma elegida: Symbian OS. Para ello se evaluarán las posibilidades que ofrece el sistema operativo, así como sus limitaciones. En esta fase se debe hacer hincapié en los aspectos más concretos que van a ser necesarios para la realización del proyecto, como lo son la multitarea, las posibilidades de comunicación y las funciones multimedia.

Una vez adquiridos los conocimientos teóricos pasaremos a la tercera fase. Ésta consiste en evaluar las posibilidades que ofrece la librería RTP escogida: la JRTPLIB, descomponiendo el trabajo de la migración en distintos bloques.

La cuarta fase consiste en la implementación de los distintos módulos, prestando especial atención al que se encarga de establecer las comunicaciones con la capa de transporte. Una vez implementados los módulos, pasaremos a realizar las modificaciones necesarias en el código de la JRTPLIB para adaptarla a su uso en Symbian OS.

Por último, en la quinta fase del proyecto se debe implementar una aplicación, que pruebe las funcionalidades de la pila RTP migrada. Adicionalmente, se pretende incluir un módulo a la aplicación que interactúe con las capas físicas de Symbian para trabajar con audio. De esta forma se pretende implementar una aplicación capaz de probar la librería mediante el envío de paquetes aislados o a través de streaming de audio.

1.3. Estructura del proyecto

En este primer capítulo se describen las motivaciones que han llevado a la realización de este proyecto y los objetivos marcados a su inicio.

En el segundo capítulo se presenta el protocolo RTP, haciendo un repaso de sus funcionalidades más importantes, así como las modelos de comunicación que permite. Se entrará en detalle de los componentes que forman RTP, prestando especial atención al protocolo de transferencia de datos y al protocolo de control RTCP.

En el tercer capítulo podremos ver, desde un punto de vista teórico y práctico, la plataforma escogida: Symbian OS. En él se detallan los aspectos más relevantes que han sido objeto de estudio para la realización del proyecto, así como una introducción a las posibilidades que ofrece este sistema operativo.

El cuarto capítulo muestra la arquitectura de la pila RTP implementada, detallando cada uno de los módulos que la forman. En este capítulo se ha utilizado notación UML para describir el diseño y explicar el funcionamiento de la pila RTP.

Por último, en el quinto capítulo se valora el estado final del proyecto, indicando los objetivos alcanzados. Así mismo, se plantean posibles ampliaciones futuras y mejoras del proyecto.

En el Anexo 1 se describen de forma teórica las capacidades multimedia que ofrece Symbian OS, prestando especial atención al streaming de audio. También veremos la implementación de un módulo de streaming. En este anexo se muestran los aspectos de diseño más importantes en la utilización de contenidos multimedia en Symbian OS.

La implementación de la aplicación que prueba la funcionalidad de todos los componentes RTP queda descrita en el Anexo 2. Así mismo se puede ver la inclusión del módulo de sonido, componiendo así una aplicación multimedia con conectividad RTP.

CAPÍTULO 2. RTP

El protocolo RTP (Real-time Transport Protocol) proporciona funciones de transporte extremo a extremo para aplicaciones que necesiten transmitir datos en tiempo real.

Este protocolo fue desarrollado por el grupo de trabajo AVT (Audio-Video Transport), un grupo de la Internet Engineering Task Force (IETF). La primera especificación de RTP fue aprobada el 22 de noviembre de 1995, dando lugar a su publicación en el RFC 1889. La primera implementación del protocolo fue de la mano de Netscape, quien a principios de 1996 anunció un framework para la transmisión de audio y video en tiempo real, basado en RTP.

Actualmente, RTP se ha convertido en el protocolo estándar de Internet para el transporte de datos en tiempo real, incluyendo audio y video. Puede ser usado tanto para medios bajo demanda como para servicios interactivos, como VoIP. La especificación publicada en el RFC 1889 ha dado paso a una revisión, que cambia ligeramente las reglas y los algoritmos para el uso del protocolo. Esta revisión ha sido publicada en el RFC 3550, así como un sinfín de perfiles para optimizar el funcionamiento del protocolo, adaptándolo a cualquier tipo de tráfico.

En este capítulo veremos un estudio teórico del protocolo RTP, destacando los aspectos más relevantes de los componentes que lo forman.

2.1. Introducción

La transmisión de medios en tiempo real a través de una red basada en IP requiere un elevado throughput, así como un retardo y jitter (variación del retardo) mínimos. En las comunicaciones en tiempo real ocasionan más problemas los altos retardos que las capacidades bajas o las pérdidas. Esto no sucede así cuando se accede a datos estáticos, como la descarga de un fichero, donde lo único que importa es el que los datos lleguen íntegramente al destino de la forma más rápida posible. Como consecuencia de esta contradicción, los protocolos pensados para el acceso a datos estáticos no se comportan bien en el envío o recepción de datos en tiempo real.

TCP es un protocolo de la capa de transporte, diseñado para proporcionar fiabilidad en entornos de ancho de banda limitado y con una alta probabilidad de error. Para cumplir esta función añade una gran cantidad de overhead, así como mecanismos que controlan el flujo para evitar las congestiones de la red. Esto supone una ralentización de la tasa de transmisión. Es por ello que para las comunicaciones en tiempo real se usan otros protocolos. UDP es el más utilizado, dado que es un protocolo muy extendido, que añade muy poco overhead, permitiendo unas tasas de transmisión muy altas. No obstante, el uso de UDP no es la solución para las comunicaciones en tiempo real, ya que no ofrece ningún tipo de fiabilidad.

El protocolo RTP se creó debido a la inexistencia de un protocolo estandarizado, apto para soportar tráfico en tiempo real sobre Internet.

La filosofía de RTP es proporcionar las herramientas necesarias a las aplicaciones para que puedan controlar las conexiones en tiempo real, permitiendo a éstas poder ofrecer una cierta calidad de servicio. Es importante destacar que RTP no proporciona calidad de servicio en la transacción de tráfico en tiempo real. La función de RTP es dotar a las aplicaciones de una cierta "inteligencia" para que puedan optimizar la transmisión de contenidos en tiempo real, adaptándola a las características y el estado de la red. RTP ha sido diseñado para que pueda cumplir estas funciones independientemente de la capa de transporte. Esto significa que RTP se utiliza como una capa intermedia entre la aplicación y el transporte. El hecho de que sea independiente a la capa de transporte permite que se pueda utilizar una gran variedad de protocolos por debajo de RTP. UDP es el protocolo más utilizado, debido a que no ofrece ningún método de control específico para acceso a contenidos estáticos (innecesarios en las comunicaciones en tiempo real) y ofrece una tasa de transmisión más elevada que, por ejemplo, TCP.

RTP es un protocolo deliberadamente incompleto, pensado para que cada aplicación pueda implementar o modificar ciertas funcionalidades, dependiendo del tipo de datos a transmitir, así como las prestaciones que se pretenden obtener. De esta forma se obtiene una especificación muy generalizada, que necesita de otros documentos para completar la especificación para su uso en una aplicación concreta:

- Perfil: Este documento define un conjunto de tipos de payload, como por ejemplo codificaciones de audio. Un perfil además puede definir extensiones y modificaciones en el protocolo RTP para ajustarse a las características de la aplicación. Existen una gran cantidad de perfiles específicos para la transmisión de distintos tipos de datos. El más común es el especificado en el RFC 3351, el cual describe las reglas para manejar comunicaciones de audio y video con muy bajo nivel de control.

- Formato del payload: En este documento se define como tiene que ser transportado por RTP un tipo particular de payload.

Normalmente nos referimos a RTP como si se tratase de un único protocolo. No obstante RTP está formado por dos componentes estrechamente unidos:

- RTP Data Transfer Protocol: Este protocolo es el que se encarga del transporte de los datos en tiempo real.

- RTP Control Protocol (RTCP): Este protocolo cumple una importante función. Se encarga de generar información de control de todos los participantes de una sesión RTP. Mediante este protocolo las aplicaciones pueden recopilar la información necesaria de todas las fuentes, con el fin de monitorizar la calidad de servicio. Gracias a la recopilación de estos datos las aplicaciones pueden tomar las

decisiones pertinentes para mantener un cierto grado de calidad de servicio.

En el siguiente esquema se muestran todos los componentes RTP, así como las capas subyacentes.

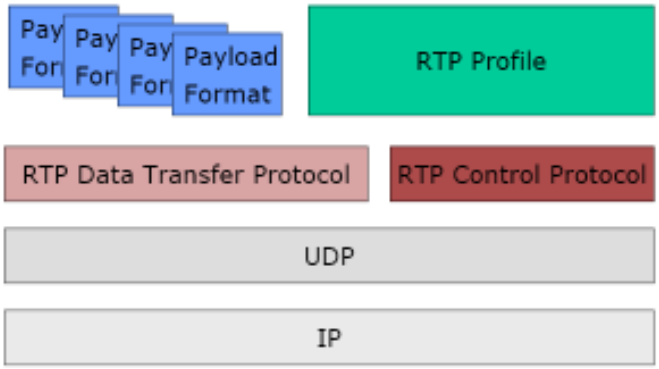

Fig. 2.1. Componentes RTP.

El conjunto de flujos RTP que es accesible por un grupo de usuarios se denomina sesión multimedia. Ésta puede estar formada por uno o varios tipos de datos. Por ejemplo, en una videoconferencia los participantes envían el audio y el video por separado. Esta división de medios se hace para que cada participante pueda descartar la recepción de algún tipo de datos, permitiendo así la adaptación de la sesión entre redes con distinta capacidad. Cada uno de estos flujos de datos recibe el nombre de sesión RTP. Por lo tanto una sesión multimedia es un conjunto de sesiones RTP.

Una sesión RTP es una asociación entre un grupo de participantes, los cuales se están comunicando con datos RTP del mismo tipo. Un participante puede estar involucrado en más de una sesión RTP a la vez, donde cada sesión transporta un medio diferente. Cada sesión RTP tiene asociado un control RTCP independiente. Para poder llevar a cabo esta separación de medios se hace necesaria la existencia de multiplexación en la capa de transporte. Para ello se asignan pares de puertos (uno para RTP y otro para RTCP) diferentes para cada sesión. Es común utilizar un puerto par para RTP y el puerto inmediatamente superior para RTCP.

RTP soporta la transferencia de datos a más de un destinatario utilizando multicast, siempre y cuando la red lo soporte. De esta forma se satisfacen las necesidades de conferencias multimedia entre uno o varios participantes, aplicación para la cual RTP ha sido diseñado primordialmente. A pesar de ser un protocolo creado con este fin, hay una gran cantidad de aplicaciones que pueden explotar la utilidad de este protocolo.

2.2. Componentes de red RTP

RTP soporta sistemas intermediarios que permiten extender las capacidades de comunicación RTP. Estos sistemas son los mezcladores y traductores. Mediante ellos se pueden establecer modelos de comunicación que permiten aumentar y optimizar la experiencia de los puntos finales, haciendo así más flexibles los servicios de contenidos en tiempo real.

Un mezclador o traductor interconecta dos o más redes distintas a nivel de transporte. Para evitar bucles al utilizar estos sistemas es necesario que estas redes utilicen diferentes protocolos de transporte, espacio de direcciones o puertos. Con el cumplimiento de uno de estos tres parámetros es suficiente. Si las redes están aisladas a nivel de red no existe el riesgo de que aparezcan bucles. La segunda medida que hay que tomar para evitar bucles consiste en no utilizar estos sistemas en paralelo, a menos que se haya hecho una distribución de las fuentes que pueden ser procesadas.

2.2.1 Mezcladores

En una sesión multimedia, se asume que todos los participantes están preparados para enviar y recibir los datos en el mismo formato. Pero se pueden dar casos en que esta condición no se cumple. Las codificaciones de los medios que son enviados se tienen que adaptar a las características de la red, de forma que todos los usuarios puedan obtener los datos correctamente en recepción. Si un solo participante accede a la sesión a través de una conexión con menos ancho de banda que el resto, todos los participantes tendrán que codificar los medios con una calidad muy inferior a la que le permite su red. Para solucionar este problema se utilizan los mezcladores.

Un mezclador recibe uno o varios flujos RTP de una o más fuentes. Este elemento puede procesar los datos de los distintos flujos para llevar a cabo distintas acciones: puede cambiar la codificación de los datos, permitiendo así adaptar los flujos RTP para que puedan ser transportados en redes con distinto ancho de banda. De esta forma se evita que todos los participantes tengan que adaptarse a las limitaciones del participante con menos capacidad. Otra función común de los mezcladores es combinar distintos flujos RTP en uno. Gracias a esto un mezclador puede, por ejemplo, recibir varios flujos de audio y combinarlos, obteniendo un único flujo a la salida. De la misma forma se pueden combinar flujos de vídeo.

Gracias a estas funciones los mezcladores permiten interconectar redes distintas, adaptando y mezclando el tráfico para adaptarlo a las características de la red. De esta forma se aumenta la escalabilidad del protocolo RTP, además de proporcionar servicios extra para determinadas subredes (creación de mosaicos con todos los flujos de vídeo, por ejemplo).

Es importante tener en cuenta que, al modificar los datos, el mezclador genera un flujo RTP con el identificador de fuente y un timestamp (ver 2.3.1.) diferentes a los de los flujos de entrada.

2.2.2. Traductores

Los traductores se utilizan para interconectar redes que no permiten una comunicación RTP directa. Esta incompatibilidad viene dada principalmente por las limitaciones que las redes imponen al uso de multicast. Los traductores pueden ser utilizados para posibilitar una sesión RTP entre redes que soporten multicast y redes unicast, pudiendo explotar los beneficios en la red multicast. Para ello se utiliza un traductor entre los dos tipos de redes. La función de éste es la de convertir cada flujo multicast en un conjunto de flujos unicast, y viceversa.

Otra posible limitación es la existencia de un cortafuegos a nivel de aplicación entre dos redes multicast, el cual esté configurado para filtrar el tráfico multicast. En este caso se pueden usar dos traductores, uno a cada lado del cortafuegos. Cada traductor encapsula los flujos multicast y los envía al otro traductor mediante unicast. A su vez el segundo traductor vuelve a recomponer el flujo original, enviándolo dentro de la red multicast.

Al contrario de lo que sucede con los mezcladores, los traductores no alteran el identificador de fuente de las sesiones RTP.

2.3. RTP Data Transfer Protocol

Este protocolo se encarga de transportar los datos en tiempo real. Para ello encapsula estos datos, por ejemplo audio o video codificados, dentro de un paquete RTP. El payload es entregado a la pila RTP directamente por la aplicación.

2.3.1. Formato del paquete RTP

A continuación se muestra el formato del paquete RTP y se define la función de cada uno de los campos de la cabecera:

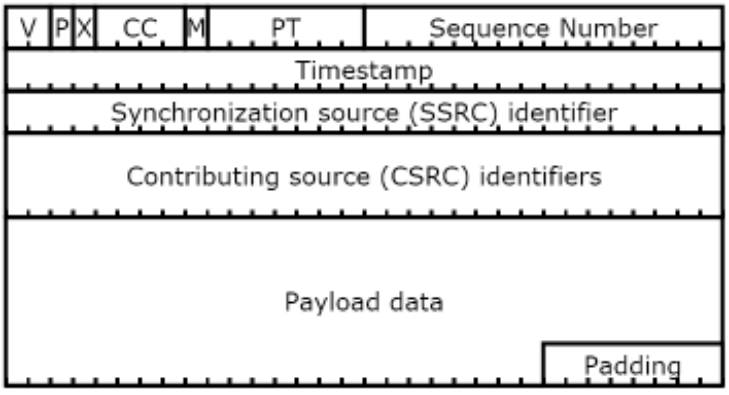

Fig. 2.2. Formato del paquete RTP.

Los primeros doce octetos son obligatorios, de modo que siempre estarán presentes en los paquetes RTP, mientras que la lista de CSRC solo aparece cuando es creada por un mezclador. Así mismo, tanto el tamaño del payload como el relleno que se muestra al final, son variables. La función de cada campo es la siguiente:

Versión (V)
Los dos primeros bits de la cabecera sirven para la identificación de la versión de RTP. Actualmente existen dos versiones. La versión más reciente (la dos) está especificada en el RFC 3550.

Padding (P)
Este campo ocupa únicamente un bit. Si éste está activo (vale 1) significa que hay uno o más octetos de relleno tras el payload. El número total de bits de padding tiene que ser múltiplo de ocho, ya que el último octeto indica el número de octetos que tienen que ser ignorados, incluyéndose éste último.

Extensión (X)
Tal y como se ha comentado anteriormente, RTP permite la extensión de sus funciones mediante los perfiles. Si este bit está activado significa que después de la cabecera RTP se inserta una cabecera de expansión. Más adelante se entrará en detalle sobre las extensiones del protocolo.

Contador de CSRC (CC)
Este campo contiene el número de identificadores CSRC que contiene la cabecera.

Marcador (M)
La función de este bit es la de avisar de posibles eventos. El significado de esta marca dependerá del perfil, facilitando a las aplicaciones la posibilidad de indicar un especial tratamiento para los paquetes que tengan este bit activado.

Tipo de payload (PT)
Este campo identifica el formato del payload. De esta forma las aplicaciones pueden saber en que formato se está codificando la imagen o el audio de la conferencia. Para ello se necesita que todas las aplicaciones participantes sigan el mismo perfil, ya que es en este documento donde se especifica el mapeo de los códigos. RTP ofrece la posibilidad de cambiar de codificación, y en consecuencia de tipo de payload, de forma dinámica mediante protocolos de control de conferencia (SIP, por ejemplo). Las aplicaciones deben ignorar los paquetes RTP que contengan tipos de payload que no puedan entender.

Número de secuencia
El número de secuencia se incrementa con cada paquete RTP enviado. El valor inicial de la secuencia se suele obtener de forma aleatoria. La función de este campo de 16 bits es permitir a la aplicación receptora detectar las pérdidas, así como poder reordenar los paquetes si es necesario.

Timestamp
Éste es uno de los campos más importantes de la cabecera. Refleja el instante en el que el primer octeto de datos ha sido muestreado. Gracias a la existencia de esta marca temporal se pueden reconstruir los mensajes en la recepción, además de sincronizar los datos provenientes de distintas sesiones RTP.

Identificador de fuente de sincronización (SSRC)
Este campo de 32 bits contiene el identificador de fuente de sincronización, SSRC en adelante. El SSRC identifica, de forma unívoca, a la fuente de un flujo de paquetes RTP. La aplicación receptora utiliza este identificador para agrupar los flujos según la fuente de sincronización, y así poder reproducirlos. De esta forma cada sesión RTP, aunque pertenezca a la misma fuente, debe tener un valor de SSRC diferente. El protocolo RTCP se encarga de asociar dos o más flujos distintos a partir de estos identificadores, como por ejemplo sincronizar el audio y el video de una misma sesión. La elección del SSRC se hace de manera aleatoria, y su procedimiento no es trivial.

Existen algoritmos complejos que se encargan de la elección, minimizando la probabilidad de colisión, ya que en una misma sesión multimedia todas las fuentes deben tener identificadores distintos. De todas formas existe la posibilidad de que se produzcan colisiones de SSRC, por lo tanto las implementaciones de RTP tienen que estar preparadas para detectar estas colisiones y tomar las medidas necesarias para solucionarlas.

Identificador de fuente de contribuyente (CSRC)
Cuando los paquetes RTP llegan a un destinatario a través de un mezclador, éste no puede saber que participantes están contribuyendo a este flujo agregado a través del SSRC, ya que identifica al mezclador. El campo CSRC es generado únicamente por los mezcladores, formando una lista con los identificadores SSRC de todas las fuentes que contribuyen a formar los paquetes de salida del mezclador. De esta forma una aplicación puede, por ejemplo saber que participantes están hablando en una conferencia de voz aunque reciban todas las fuentes mezcladas. El número de fuentes que contiene la lista CSRC viene especificado en el campo CC de la cabecera. Este campo es de 4 bits, lo que implica que como máximo la lista puede contener 15 SSRC, con 32 bits cada uno. Por lo tanto la longitud de este campo oscila entre 0 y 480 bits.

2.3.2. Modificaciones de la cabecera RTP

Como se ha comentado anteriormente, RTP es un protocolo maleable. Las aplicaciones tienen distintas posibilidades para personalizar las funciones que puede ofrecer la cabecera RTP.

La personalización más sencilla que las aplicaciones pueden hacer sobre la cabecera es el uso de los campos marcador y tipo de payload. Éstos vienen en la cabecera estándar de RTP, ya que son dos funciones muy beneficiosas para la mayoría de las aplicaciones. Al estar dentro de la cabecera estándar se evita

que las aplicaciones tengan que implementarlas. Además de esta forma se aumenta la eficiencia del protocolo, ya que si las aplicaciones tuviesen que implementar estos campos, se tendría que utilizar una palabra de 32 bits, de los cuales una gran parte tendrían que ser de relleno.

En algunos casos es necesaria información extra, relativa a un tipo concreto de payload, como una codificación poco común de video. Cuando el campo destinado a identificar el tipo de payload no es suficiente se puede transportar información en el campo de payload, añadiendo una cabecera justo al principio de éste.

Por último RTP está preparado para soportar extensión de cabecera. Como se ha comentado en el apartado anterior, si el bit X está activado significa que justo antes del campo de payload hay una extensión de cabecera. Este mecanismo es utilizado por aplicaciones que requieren funciones especiales para un tipo de payload concreto. RTP permite una única extensión de cabecera, siendo ésta de tamaño variable y definida por un perfil concreto. El formato de esta extensión se muestra a continuación.

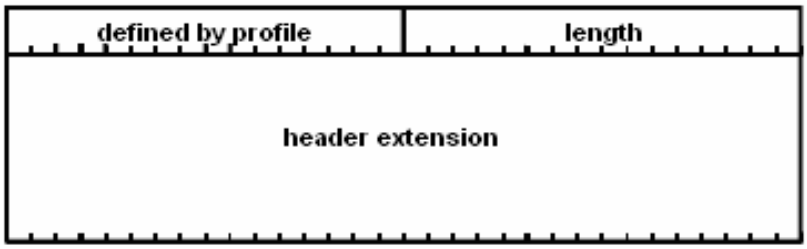

Fig. 2.3. Formato de la extensión de cabecera RTP.

Como se puede observar la extensión cuenta con dos campos de 16 bits fijos, seguidos de un campo variable, en el que la aplicación deberá especificar los campos que crea necesarios. El primer campo está reservado para que sea definido por el perfil. De esta forma una aplicación puede, por ejemplo, utilizarlo para diferenciar entre distintos tipos de extensión. El segundo campo indica la longitud en palabras de 32 bits de la extensión de la cabecera, sin contar la primera palabra, en la que están contenidos los dos campos fijos.

El hecho de que RTP permita estas extensiones es muy beneficioso, ya que los requerimientos de las comunicaciones en tiempo real están en constante evolución. Si no se permitiesen posibilidades de extensión, el protocolo correría el riesgo de quedar obsoleto rápidamente.

2.4. RTP Control Protocol (RTCP)

La función del protocolo RTCP es la difusión de mensajes de control de forma periódica. Estos mensajes pueden ser utilizados por las aplicaciones para conocer el estado de la comunicación en todos los puntos finales de una sesión

multimedia, pudiendo así adaptar las transmisiones para conseguir una cierta calidad de servicio. Además, gracias a RTCP se pueden sincronizar varios streams pertenecientes a distintas sesiones RTP.

2.4.1. Funciones de RTCP

Al igual que ocurría con el protocolo RTP, RTCP también puede ser modificado para adaptarse a aplicaciones concretas. Se pueden crear diferentes tipos de paquetes RTCP independientes a la especificación del protocolo RTP, pero todas las implementaciones de RTCP tienen que cumplir las siguientes cuatro funciones:

1. La función principal de RTCP es proporcionar feedback entre los distintos componentes de una sesión multimedia. Esto permite que se pueda monitorizar la calidad de la distribución de los datos en todos los extremos de la comunicación. Para ello, tanto los emisores como los receptores, generan paquetes con información relativa al estado de la comunicación, como número de paquetes perdidos, retardos y niveles de jitter, por ejemplo. Estos datos pueden ser directamente utilizados para el cálculo de codificaciones variables, creando inteligencia en los extremos, permitiendo así controlar la degradación del servicio por congestión en la red. Esta función es llevada a cabo a través de dos tipos de paquete RTCP (sender report y receiver report), que veremos más adelante. Si la red utiliza un mecanismo de distribución multicast, estos mensajes pueden llegar a aplicaciones conocidas como monitores. Estos monitores pueden encargarse de controlar el estado de las comunicaciones y hacer notificaciones a las aplicaciones si es necesario. De esta forma se puede liberar a las aplicaciones de realizar esta tarea.

2. RTCP es el encargado de transportar el nombre canónico (CNAME) de las fuentes RTP. Éste es un identificador persistente a nivel de transporte. Su existencia es necesaria, debido a que los identificadores SSRC pueden sufrir variaciones en presencia de colisiones, así como por reinicio en las aplicaciones. Los receptores necesitan tener un CNAME asociado a cada fuente para poder identificar a los participantes. Como se ha explicado en el apartado anterior, el SSRC identifica una sesión RTP. Por lo tanto, una fuente que esté transmitiendo, por ejemplo, audio y video en diferentes sesiones RTP estará utilizando dos SSRC diferentes. Para poder sincronizar estas dos sesiones en la recepción, la aplicación receptora necesita el CNAME de la fuente para poder asociar las distintas sesiones RTP de una única fuente. De esta forma RTCP permite hacer una asociación entre uno o varios SSRC y un CNAME.

3. Las implementaciones de RTCP deben controlar la tasa de envío de paquetes. De esta forma se puede obtener un protocolo escalable, capaz de soportar un gran número de participantes. Debido a que todos los participantes tienen que enviar mensajes de control (tanto los

receptores como los emisores) cada aplicación puede conocer el número total de participantes. Este número debe ser usado para establecer un control en la tasa de envío de paquetes RTCP.

4. Esta última función es opcional. Se trata de una mínima información de control, destinada a su uso para las aplicaciones. Gracias el envío de paquetes que contienen información de los participantes, las aplicaciones pueden, por ejemplo, mostrar sus datos por pantalla. No es la intención del protocolo gestionar este tipo de información a gran escala, ya que para ello existen protocolos de más alto nivel, como SIP.

2.4.2. Formato del paquete RTCP

Existen diferentes tipos de paquetes RTCP. En la especificación del protocolo RTP se definen cinco tipos distintos. Con ellos se pueden llevar a cabo las funcionalidades mínimas, expuestas en el apartado anterior. Estos tipos de paquete son los siguientes:

- Sender Report (SR): Informes creados por las fuentes que envían flujos de paquetes RTP. Sirven para monitorizar el estado de la sesión multimedia desde el punto de vista de los emisores.

- Receiver Report (RR): Éstos son informes creados por las fuentes que reciben paquetes RTP. Cumplen la misma función que los SR, pero desde el punto de vista de los receptores.

- Descriptor de fuente (SDES): Transporta parámetros para describir a las distintas fuentes, principalmente el nombre canónico CNAME.

- BYE: Indica el fin de una participación.

- APP: Este paquete transporta las funciones específicas de aplicaciones concretas.

Más adelante se detallarán estos tipos de paquete, ya que son estándares en cualquier implementación de RTP.

Todos paquetes van precedidos por una cabecera fija, parecida a la de RTP. A continuación de esta cabecera se añaden los campos propios de cada tipo de paquete. Es imprescindible que el paquete resultante se múltiplo de 32 bits.

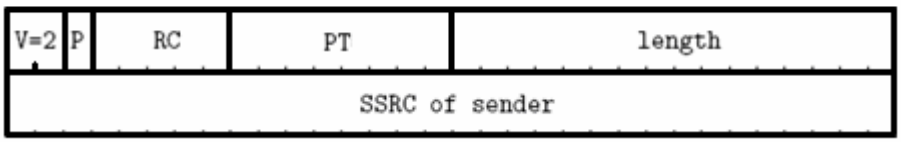

Fig. 2.4. Cabecera fija de los paquetes RTCP

Se puede observar que en esta cabecera fija aparecen los siguientes campos:

Versión (V) y padding (P)
Se trata de los mismos campos que aparece en los paquetes RTP. Especifican la versión del protocolo RTP que se está utilizando y si existen bits de relleno.

Campo variable
El tercer campo de 5 bits contiene información relativa al tipo de paquete. Normalmente se utiliza como contador.

Tipo de paquete (PT)
Este campo identifica el tipo de paquete RTCP.

Longitud
Indica la longitud total del paquete RTCP, incluyendo la cabecera. Al contrario de lo que sucede con RTP, en RTCP este campo es necesario, ya que, como se verá a continuación, un paquete de la capa de transporte contendrá más de un paquete RTCP.

SSRC del emisor
Contiene el identificador de fuente de sincronización del emisor que ha creado el paquete RTCP.

Los paquetes RTCP son muy pequeños, ya que no transportan ningún tipo de payload. Esto produce que, al ser encapsulado por las capas subyacentes se obtenga una eficiencia muy baja. Para aumentar esta eficiencia se obliga a que los paquetes RTCP sean enviados en grupos variables dentro de un solo paquete de la capa de transporte (por ejemplo UDP). A esta agrupación de paquetes se le denomina paquete compuesto. No existe un número concreto de paquetes para la formación del paquete compuesto, ya que éste viene determinado por la capacidad de la capa de transporte, ni tampoco especificaciones sobre el orden de dichos paquetes. Cada paquete RTCP puede ser procesado por los elementos RTP de la red de forma independiente.

Aunque no exista un orden concreto en la formación del paquete compuesto, la especificación de RTP impone los siguientes criterios:

- Todos los paquetes compuestos deben incluir informes de recepción o de envío (RR o SR). Estos mensajes deben ser enviados con la mayor cadencia posible, de forma que las estadísticas sean lo más fidedignas posible. Todos los paquetes compuestos deben empezar con uno de estos informes.

- Cuando se incorpora un nuevo receptor a una sesión multimedia es imprescindible que obtenga los nombres canónicos de los emisores, ya que hasta que no conoce los CNAME no puede sincronizar distintas sesiones RTP. Por ello en cada paquete compuesto debe haber un paquete descriptor de fuente.

Por lo tanto, cada paquete compuesto debe incorporar, como mínimo dos paquetes RTCP. Cada participante puede enviar un único paquete compuesto por cada intervalo de informe. De esta forma se puede hacer correctamente la estimación del ancho de banda asignado a cada participante. Si el número de paquetes compuestos fuese arbitrario, no se podría tener constancia de la cantidad de usuarios en la sesión multimedia, lo que imposibilitaría el cálculo del ancho de banda de la sesión. Si la cantidad de participantes es muy grande, y en un único paquete de la capa de transporte no se pueden incluir todos los informes de recepción únicamente se enviará una selección de éstos.

2.4.3. Tipos de paquete RTCP

A continuación se describen las principales funcionalidades de los cinco tipos de paquete RTCP estándar.

2.4.3.1. Informes de envío y recepción (SR y RR)

Los informes de envío y recepción contienen los mismos campos. La única diferencia, a parte del código de tipo de paquete, se encuentra en que el informe de envío tiene 20 bytes extras para ofrecer información sobre el emisor que ha generado el paquete.

Estos informes contienen bloques con información sobre la recepción de cada una de las sesiones RTP. Además se permiten extensiones, creando un bloque al final del paquete RTCP. Los campos de este bloque pueden ser libremente incluidos por las aplicaciones, especificándolos en el documento de perfil.

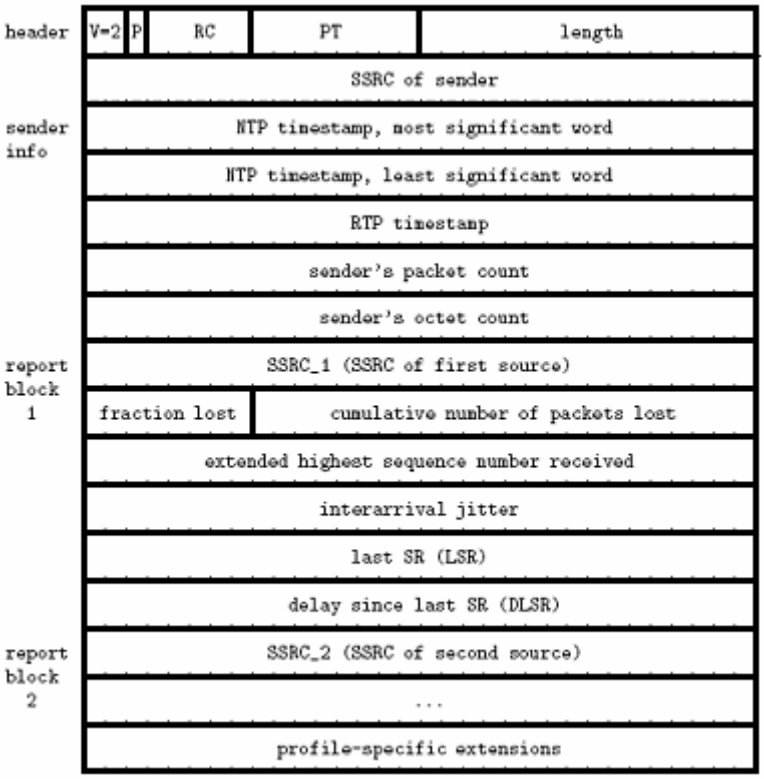

Fig. 2.5. Formato del paquete Informe de envío y recepción

Los primeros 20 bytes que siguen a la cabecera son propios únicamente de los informes de envío. Se puede observar también la existencia de la cabecera fija, detallada anteriormente. A continuación se detallarán los campos exclusivos de los informes de envío, que resumen los envíos de datos del transmisor.

NTP timestamp
Este campo ocupa dos palabras de 32 bits. Indica el tiempo real en el que el paquete ha sido generado. Al recibir las respuestas de los receptores mediante los informes de recepción, se puede calcular el retardo de propagación de la red comparando.

RTP timestamp
Contiene la misma marca temporal, pero en las mismas unidades y con el mismo offset que en los paquetes de datos RTP.

Contador de paquetes enviados
Indica el total de paquetes RTP que el emisor ha enviado hasta la creación del informe.

Contador de octetos enviados
Indica el número total de octetos que el emisor ha enviado. Solamente se cuentan los bits del campo de payload del paquete RTP.

La tercera sección de estos informes es común para ambos. Se trata de un número variable de bloques, cada uno asociado a una sesión RTP activa. Cada

uno de estos bloques contiene estadísticas de la recepción el la fuente que lo genera. Para ello se definen los siguientes campos en cada bloque:

SSRC
Identifica a la fuente a la cual se refieren las estadísticas del bloque.

Fracción perdida
Indica la fracción de paquetes RTP que han sido perdidos desde el envío del anterior informe. Este número solo hace referencia a la sesión RTP de la que se está informando, no al número de paquetes perdidos en la sesión multimedia.

Número acumulativo de paquetes perdidos
Este paquete indica el número total de paquetes RTP perdidos desde el establecimiento de la sesión RTP. Al igual que el campo anterior, solamente se hace referencia a la sesión de la que se informa. El hecho de poseer un contador acumulativo permite que se puedan hacer estimaciones de los paquetes perdidos entre diferentes periodos de tiempo, no solo entre dos informes consecutivos o en el total de la sesión.

Extensión del número de secuencia más alto
Este campo de 32 bits se divide en dos partes. En los últimos 16 bits se indican el número de secuencia del último paquete de datos RTP recibido de la fuente. Los primeros 16 bits son una extensión de ese número de secuencia. Esta extensión cumple funciones de control de errores.

Intervalo de jitter
Este campo contiene una estimación del jitter entre llegadas de paquetes de datos RTP. Este cálculo se realiza en función de la marca temporal de envío y el instante de tiempo en que se reciben los paquetes. El cálculo se hace en base a los dos últimos paquetes recibidos. La monitorización de este parámetro es importante, ya que puede indicar la existencia de principios de congestión en la red, antes de que dicha congestión se traduzca en un aumento de la tasa de pérdidas. De esta forma, si se toman las medidas preventivas necesarias ante el aumento del jitter se puede llegar a prevenir la pérdida de paquetes.

Último informe de envío (LSR)
Contiene el timestamp del último paquete RTCP de informe de envío recibido de la sesión RTP.

Retardo desde el último informe de envío (DLSR)
Este campo indica el tiempo que ha pasado desde la recepción del último informe de envío.

Por último, en la figura se puede observar la existencia de una cuarta sección, que es opcional. En esta sección se permite a las aplicaciones extender las funcionalidades de los informes de envío y recepción, de forma que se puedan incluir más datos que permitan la creación de diferentes estadísticas. Debido a que cada paquete compuesto debe contener un informe de envío o recepción,

esta sección de extensión puede ser utilizada por las aplicaciones para definir un nuevo tipo de paquete RTCP.

Al procesar todos estos datos se pueden obtener una gran cantidad de parámetros útiles para monitorizar la calidad de servicio. Se pueden calcular tasas de pérdidas, desviaciones del retardo, tasa de envío de las fuentes y un largo etcétera, tomando intervalos de tiempo pequeños o sobre el tiempo total de la sesión multimedia.

2.4.3.2. Descriptor de fuente (SDES)

Este paquete está formado por la cabecera estándar y un conjunto de bloques de número variable. En cada uno de esos bloques se incluye información descriptiva de una sesión RTP.

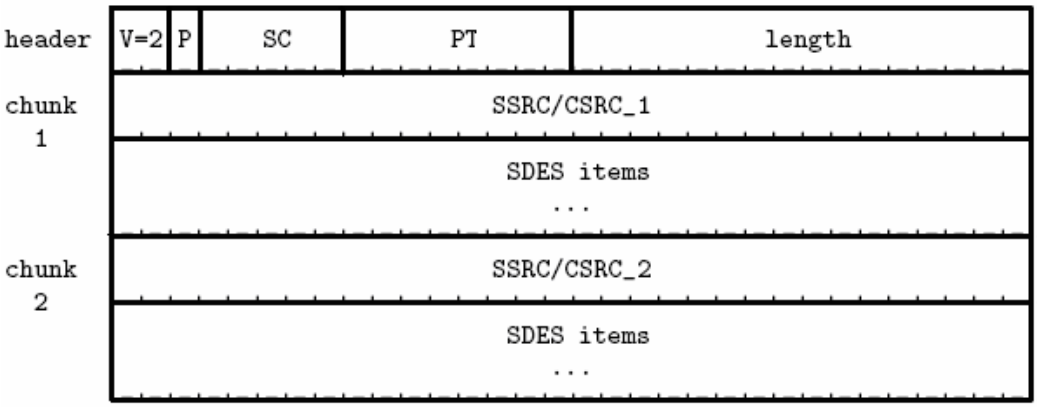

Fig. 2.6. Formato de paquete descriptor de fuente.

Cada bloque consta de dos partes. En primer lugar se indica a que sesión o sesiones RTP hacen referencia las descripciones. La segunda parte es variable. Su contenido es diferente según el tipo de descripción que se haga de la fuente. Consta de los siguientes campos:

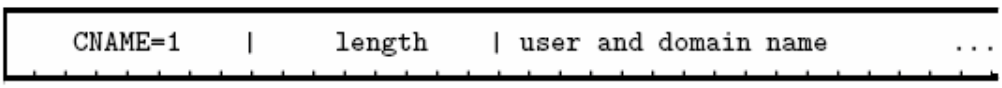

Fig. 2.7. Campos de SDES items.

En primer lugar se utilizan 8 bits para identificar el tipo de descripción. A continuación se indica la longitud del tercer campo (no se incluyen estos dos octetos). Este tercer campo contiene la descripción de la fuente. Su contenido está codificado en formato de texto para poder albergar cualquier tipo de descripción, ya que sería imposible mapearlas en códigos.

La especificación del protocolo RTP define ocho tipos distintos de descripciones:

- CNAME: Contiene el identificador canónico de la fuente RTP. Este es el tipo de descripción más importante. Tanto es así que es el único tipo de descripción obligatorio en cualquier implementación de RTP.

- NAME: Nombre real del participante, utilizado para identificar a la fuente a través de la interfaz de las aplicaciones.

- EMAIL: Dirección de correo electrónico del participante.

- PHONE: Número de teléfono del participante.

- LOC: Indica la localización geográfica del participante.

- TOOL: Un string que indica la versión de la aplicación que genera el flujo RTP.

- NOTE: Esta descripción permite a los usuarios informar sobre su estado.

- PRIV: Este string se utiliza para que las aplicaciones puedan incluir sus propias descripciones.

Como se ha comentado anteriormente, en cada paquete compuesto RTCP debe haber un descriptor de fuente. Este SDES debe contener al menos un bloque con el tipo de descriptor CNAME.

2.4.3.3. BYE

Cuando un participante desea abandonar la sesión transmite un paquete BYE, con el fin de informar a los otros participantes.

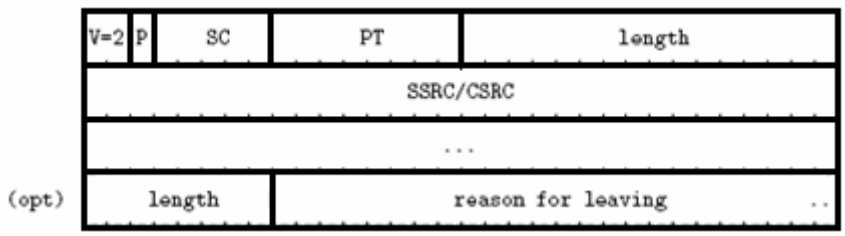

Fig. 2.8. Formato del paquete BYE.

El paquete BYE está formado por la cabecera RTCP estándar y el identificador SSRC del participante o participantes que abandonan la sesión. Opcionalmente

se puede añadir un campo en el que se explican los motivos del abandono, precedido por un campo que indica la longitud del mensaje.

Es muy común que la mayoría de los usuarios abandonen la sesión de forma simultánea, por ejemplo al terminar una reunión o al finalizar de la visualización de contenidos en directo. Si el número de participantes es muy alto (más de 50) se corre el riesgo de sufrir una inundación de paquetes BYE en la red. Para evitar esto los participantes deben seguir un algoritmo de contención, que les permite ir abandonando la sesión multimedia de forma escalonada. Si el número de participantes es inferior a 50 se puede enviar directamente un paquete BYE, sin tener que seguir ningún algoritmo. Con este mecanismo se asegura que, en el peor de los casos, el porcentaje de ancho de banda de la sesión utilizado por paquetes RTCP no supere el 10% (5% para otros paquetes RTCP y 5% para paquetes BYE).

2.4.3.4. APP

Este tipo de paquete RTCP queda a disposición de las aplicaciones. Éstas pueden utilizarlo para incluir funciones específicas sin tener que registrar nuevos tipo de paquete, lo que supone una facilidad para los desarrolladores, a la vez que se estandarizan los formatos.

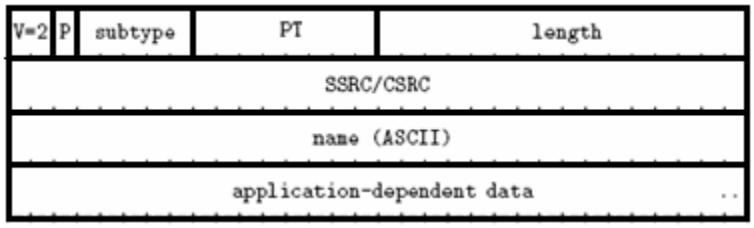

Fig. 2.9. Formato del paquete APP.

El paquete APP contiene, además de la cabecera estándar, un campo con un texto codificado en ASCII, el cual identifica el tipo de paquete APP del que se trata. A continuación se utiliza un campo variable, en el que se insertarán los diferentes campos que formarán el paquete específico de la aplicación.

CAPÍTULO 3. Symbian OS

Los requisitos que el mercado de telefonía móvil exige a los terminales han cambiado mucho en muy poco tiempo. Hace solo unos años un terminal móvil era únicamente un teléfono que podía usarse para mantener conversaciones de voz, independientemente del lugar en el que se encontraba el usuario, utilizando una red inalámbrica. Actualmente parece que este concepto se está extrapolando a la informática de sobremesa. Los usuarios demandan nuevas prestaciones a los terminales, haciendo que éstos dejen de ofrecer únicamente servicios tradicionales de voz y empiecen a integrar capacidades multimedia, de datos y las nuevas tecnologías de comunicación inalámbrica.

Estas nuevas capacidades han hecho que los terminales se conviertan en sofisticados dispositivos, con la necesidad de un sistema operativo robusto y eficiente, capaz de gestionar todos los servicios demandados. Tradicionalmente, los fabricantes han creado sus propios mini sistemas operativos, dedicando años de desarrollo y obteniendo un sistema propietario, con unas aplicaciones propias de cada fabricante. Estos mini sistemas operativos se han ido adaptando para satisfacer las demandas de los usuarios, ofreciendo, por ejemplo reproductores de contenidos multimedia.

El problema de los sistemas propietarios surge cuando un amplio sector de usuarios demanda terminales con capacidades similares a las que puede ofrecer una PDA, capaces de soportar cualquier estándar y que permita la instalación de aplicaciones complejas, robustas y fiables desarrolladas por terceras partes independientes. Bajo estas premisas se hace imprescindible el abandono de los sistemas operativos propietarios en favor de una nueva generación de sistemas operativos para móviles, capaces de estandarizar los servicios ofrecidos por los terminales, así como ofrecer un entorno de programación abierto y de gran potencial. Con la intención de satisfacer estas nuevas demandas nació el sistema operativo Symbian OS.

Symbian se formo como empresa en Junio de 1998, siendo propiedad de las más importantes empresas del sector. Entre sus accionistas se encuentran Sony Ericsson, Nokia y Siemens. Además otras empresas importantes poseen licencias de Symbian OS, como Motorola por ejemplo. Siendo concebido con el respaldado por los fabricantes, Symbian OS se encuentra actualmente a la cabeza del mercado de sistemas operativos para “smartphones”, encontrando en la marca finlandesa Nokia su mayor baluarte. Tanto es así que algunas de las marcas que impulsaron y se involucraron en la creación del sistema operativo se empiezan a desvincular de Symbian, optando por otras soluciones, como por ejemplo Linux.

La filosofía de Symbian engloba los aspectos más relevantes que el mercado de telefonía móvil demanda. Esta filosofía es descrita por Symbian a través de cinco puntos clave:

- Teléfonos móviles pequeños y móviles: Los teléfonos móviles tienen que ser pequeños y, por definición, móviles. Esto implica que el

terminal tiene que estar siempre disponible, ofreciendo una rápida respuesta, así como disponer de una batería que permita horas de independencia. Para ello Symbian OS ha sido diseñado para encontrar el equilibrio entre potencia y eficiencia energética, ofreciendo una gran capacidad de proceso en muy poco espacio y con el mínimo consumo de energía posible.

- Diseño orientado a un mercado diversificado: Symbian OS ha sido diseñado para satisfacer las necesidades de varios sectores del mercado de telefonía móvil. La intención de Symbian es dirigirse a un mercado masivo formado por consumidores, empresas y profesionales.

- Soporte de conexión: La conectividad requiere un sistema multitarea, prestaciones suficientes para soportar comunicaciones en tiempo real y un amplio conjunto de protocolos de comunicación. Symbian OS cumple con estos requisitos. Además admite la creación de pilas de protocolos por desarrolladores independientes, de forma que puede llegar a soportar cualquier tipo de conexión.

- Diversidad de productos: Los desarrolladores de software desean programar para una única plataforma estandarizada, mientras que los fabricantes pretenden tener una gran cantidad de productos únicos, que les proporcionen distinción con respecto a sus competidores. Symbian OS puede solucionar esta contradicción. La interfaz gráfica del sistema operativo queda separada del núcleo del sistema, dejando libertad a los fabricantes para que la implementen según sus necesidades. De esta forma cada fabricante puede innovar sin restricciones, desarrollando dispositivos que pueden ir desde terminales convencionales hasta dispositivos con una gran pantalla táctil y teclado completo.

- Plataforma abierta: Un sistema operativo enfocado a un mercado masivo tiene que permitir el desarrollo de terceras partes. Symbian OS proporciona documentación, a través de su página web y de Nokia, así como SDKs y herramientas. Los desarrolladores pueden implementar aplicaciones y servicios mediante C++ y Java de forma nativa. Symbian cuenta con una amplia variedad de APIs destinadas a los desarrolladores independientes.

En este capítulo se intenta ofrecer una visión global de la arquitectura de Symbian OS, así como una explicación de las principales APIs que han sido objeto de estudio para este proyecto.

3.1. Arquitectura de sistema

El kernel de Symbian OS forma un compacto sistema operativo multitarea, con muy poca dependencia de los periféricos. Los accesos al hardware únicamente pueden ser realizados en modo privilegiado. Las aplicaciones de Symbian OS

trabajan siempre en modo usuario, por lo que no pueden acceder directamente al hardware. El kernel pone los servicios del hardware a disposición de las aplicaciones a través de APIs. Cuando las aplicaciones desean interactuar con el hardware del dispositivo tienen que hacer una petición a través de una interfaz, de modo que éstas se comportan como clientes, mientras que las APIs del sistema operativo actúan como servidores. Symbian OS ha sido diseñado bajo este patrón cliente servidor y en el uso de eventos.

Normalmente, las aplicaciones de Symbian OS suelen tener métodos que permanecen a la espera de eventos generados por el propio sistema operativo, o por los dispositivos de entrada. Las aplicaciones suelen estar formadas por un módulo que se encarga de capturar eventos y uno o varios módulos para procesar estos eventos y realizar las acciones computacionales de la aplicación. Estos módulos reciben el nombre de *engine*.

El uso de esta arquitectura cliente servidor y la captura de eventos han sido diseñados para optimizar las prestaciones de los dispositivos, haciendo poco uso de la memoria y obteniendo una gran velocidad de interacción entre los diferentes niveles hardware y software.

3.2. Multitarea

La multitarea es una técnica utilizada por los sistemas operativos para compartir un único procesador entre varios procesos independientes que se estén ejecutando a la simultáneamente.

El proceso es la unidad fundamental de protección en Symbian OS. Cada proceso tiene su propio espacio de memoria. De esta forma la memoria de escritura no puede ser compartida entre diferentes procesos.

La unidad fundamental de ejecución en Symbian OS es el thread. Un proceso puede tener uno o varios threads. Cada thread ejecuta una tarea de forma totalmente independiente a los demás. Éstos pueden compartir la memoria, de forma que no están tan aislados como los procesos. Este hecho conlleva una especial atención para no sobrescribir zonas de memoria que estén siendo utilizadas por otro thread (zonas de exclusión mutua).

Existen dos tipos de multitarea, la apropiativa y la no apropiativa. La diferencia entre estos dos tipos reside en quien decide el momento de cambiar de tarea. En el tipo apropiativo el cambio se fuerza a través de un proceso de sistema llamado scheduler. Este proceso se encarga de suspender la tarea que se está ejecutando y dar paso a otra. Este cambio se realiza cada vez que una tarea ha agotado un tiempo de ejecución que tenía fijado, llamado “time slice”.
Por otra parte, en la multitarea no apropiativa es la propia tarea que se está ejecutando la que decide cuando deja libre el procesador. En ambos casos la conmutación tiene que estar gestionada por un scheduler, que dé permiso a una cierta tarea para que pueda continuar su ejecución.

3.2.1. Multitarea en Symbian OS

Symbian OS puede ejecutar varias aplicaciones y servidores simultáneamente. Para ello utiliza, al igual que Windows NT o Unix, la multitarea apropiativa para gestionar el cambio de contexto entre los diferentes threads. Un sistema operativo no puede dejar que las aplicaciones gestionen el uso del procesador, ya que así una tarea podría obtener el procesador y no liberarlo hasta que terminase su ejecución. Symbian OS implementa además multitarea no apropiativa en el contexto de cada thread.

Por lo tanto el scheduler del sistema operativo tiene el control sobre un cierto número de tareas, a las cuales otorgará un tiempo de ejecución repetidamente. A su vez cada una de estas tareas podrá gestionar su tiempo de ejecución, creando subtareas. Éstas no estarán gestionadas por el sistema operativo, sino por la propia tarea (multitarea no apropiativa).

Mediante los Active Objects y el Active Scheduler, Symbian OS proporciona un framework que permite implementar multitarea no apropiativa.

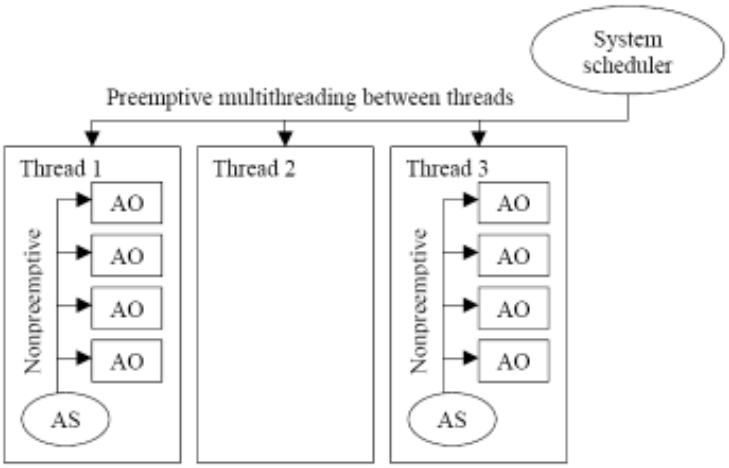

Fig. 3.1. Multitarea apropiativa y no apropiativa de Symbian OS

Cada thread puede tener un Active Scheduler que se encargue de planificar el tiempo de ejecución, repartiéndolo entre diferentes Active Objects. El Active Scheduler implementa un mecanismo no apropiativo, por lo tanto un Active Object no será interrumpido durante su ejecución. Éste será el que retornará el control al Active Scheduler una vez haya finalizado su tarea.

Arquitectura cliente/servidor y métodos asíncronos

El micro kernel de Symbian OS está basado en una arquitectura cliente/servidor. Frecuentemente, las aplicaciones son clientes que usan recursos del sistema a través de servidores. Normalmente se proporciona una clase que hace de proxy entre la aplicación y el servidor que tiene acceso a los

recursos. En la nomenclatura de Symbian estas clases siempre empiezan por R mayúscula (R-clases).

Las R-clases tienen dos tipos de métodos:

- Asíncronos: Cuando el thread del cliente llama a uno de estos métodos, la R-clase construye un mensaje y lo envía al servidor. En cuanto el mensaje es enviado se continúa la ejecución del cliente, sin esperar una respuesta por parte del servidor.
  Por otra parte, el servidor tiene un thread que recibe los mensajes, mediante el cual lee los datos enviados y procesa la petición. Cuando la petición se ha completado el servidor lo notifica al cliente y le envía un código con el resultado.
  Cuando se recibe la notificación en el thread del cliente, ésta tiene que ser procesada.

- Síncronos: El mecanismo seguido al llamar a un método síncrono es muy similar al asíncrono. La principal diferencia entre ambos es que, en el caso síncrono, cuando el cliente llama al método de la R-clase no continúa con su ejecución. El thread del cliente queda bloqueado hasta que se recibe la notificación por parte del servidor de que la petición ha sido procesada.

La utilización de métodos asíncronos comporta una dificultad añadida. El cliente que llama a funciones asíncronas tiene que recordar cada llamada de este tipo y recorrerlas cuando se produzca una notificación del servidor. En caso de utilizar métodos síncronos no existe esa dificultad, ya que la respuesta se puede procesar fácilmente, debido a que el thread del cliente se bloquea al hacer la llamada y se retoma cuando el servidor envía la notificación.

Pero se pueden ver dos ventajas en la utilización de métodos asíncronos:

- En primer lugar la utilización de estos métodos supone una mayor eficiencia, ya que el thread del cliente podrá hacer otras funciones mientras la petición se está procesando en el servidor.

- La segunda ventaja es la percepción del usuario. Si una petición síncrona no es procesada inmediatamente por el servidor el thread del cliente puede quedar bloqueado el tiempo suficiente para que el usuario perciba que el terminal no responde. En cambio al utilizar métodos asíncronos los eventos generados por el usuario podrán ser atendidos aunque una o varias peticiones estén a la espera de ser procesadas.

Con el fin de simplificar la tarea del programador, Symbian OS proporciona un framework que hace más sencilla la utilización de servicios asíncronos: los Active Objects y el Active Scheduler.

3.2.2. Active Objects y el Active Scheduler

La principal motivación del uso de Active Objects es proporcionar un método sencillo de proceso de eventos, de forma que se evite tener que ejecutar un thread para cada fuente de eventos. Hay varios motivos que justifican la elección de Active Objects para el tratamiento de eventos en Symbian:

- La comunicación entre threads es lenta y requiere la implementación por parte del programador. Usando Active Objects no existe este problema, ya que éstos se ejecutan en un mismo thread, por lo tanto no se necesita ningún mecanismo especial para compartir los datos.

- La conmutación entre threads está controlada por el sistema, lo que supone un consumo de CPU.

- El acceso a zonas de exclusión mutua tiene que estar controlado si se está compartiendo memoria entre distintos threads. Esto se lleva a cabo mediante semáforos, que añaden complejidad al código y disminuyen el rendimiento. Con Active Objects no se pueden producir conflictos en estas zonas de memoria. Al usar multitarea no apropiativa no se puede dar el caso de dos Active Objects accediendo a la vez a la misma zona de memoria.

Todos estos motivos tienen en común reducir la carga del procesador. Hay que tener en cuenta que el procesador es un bien escaso en los terminales móviles.

Active Objects

Recibe el nombre de Active Object cualquier clase que derive de `CActive`. La clase `CActive` tiene dos métodos puramente virtuales, lo que significa que obligatoriamente tendrán que ser implementados por el programador en el Active Object. Estos dos métodos son `RunL()` y `DoCancel()`.

El más importante es el método `RunL()`. Éste será llamado por el Active Scheduler cuando el servidor haya procesado la petición asíncrona. Por lo tanto en el código de este método se puede procesar el resultado de la petición.

Otro elemento a tener en cuenta de los Active Objects es la variable `iStatus`, que tiene que ser del tipo `TRequestStatus`. Esta variable tiene que ser pasada como argumento a todos los métodos asíncronos. La función de esta variable es importante, ya que a través de ella el servidor puede notificar al Active Scheduler que una petición se ha procesado.

De forma simplificada, una aplicación con Active Objects sigue los siguientes pasos:

1. Se crea el Active Object y se añade al Active Scheduler. Los pasos a seguir para este punto se describen más adelante.

2. El Active Object hace una llamada a un método asíncrono, pasándole como argumento la variable `iStatus`. Este método le da a la variable el valor `KRequestPending`, para indicar que el Active Object está a la espera de que se procese una petición. A continuación el método asíncrono crea un mensaje y lo envía al servidor. Por último el método asíncrono retorna. Acto seguido el Active Object tiene que llamar al método `SetActive()`, el cual indica al Active Scheduler que debe añadirle a su lista de Active Objects con peticiones pendientes.

3. Una vez que el servidor ha terminado de procesar la petición, cambia el valor de la variable `iStatus`, introduciendo el código resultante de la petición.

4. Cuando el Active Scheduler comprueba que la variable `iStatus` de un Active Object tiene un valor distinto a `KRequestPending`, éste llama al método `RunL()` del Active Object.

5. En el método `RunL()` se procesa el resultado de la petición. Una práctica habitual es que en este método se haga una nueva petición asíncrona y se llame al método `SetActive()`. De esta forma se podrá estar a la espera de eventos de manera ininterrumpida.

La tarea que esté realizando el Active Object tiene que poder ser cancelada. Al derivar de la clase `CActive` todos los Active Objects tienen un método llamado `Cancel()`. Cuando este método es llamado el Active Scheduler automáticamente llama al método `DoCancell()`. Como se ha dicho anteriormente, este método es virtual, lo que significa que tendrá que ser implementado por el programador. Esta función debe contener el código para cancelar todas las peticiones asíncronas que se estén procesando. Normalmente las clases con métodos asíncronos tienen otros métodos que permiten cancelar las peticiones que se están procesando. Cuando el servidor ha cancelado la petición lo indica al Active Scheduler, dando el valor `KErrCancel` a la variable `iStatus`.

Active Scheduler

Todas las aplicaciones con interfaz gráfica de Symbian son un proceso que tiene, al menos, un thread principal. Este thread tiene instalado un Active Scheduler, de forma que podrá utilizar Active Objects. En cambio, cualquier thread que sea creado por el programador no tendrá instalado el Active Scheduler. A continuación se muestran los pasos que hay que seguir para que se pueda gestionar un Active Object:

1. Se crea y se instala el Active Scheduler en el thread, mediante el método `CActiveScheduler::Install()`. Este paso solo es necesario si se trata de un thread que no tenga previamente instalado un Active Scheduler.

2. Se añade el Active Object al Active Scheduler, utilizando el método `CActiveScheduler::Add()`.

Aspectos de diseño

Es importante conocer bien el comportamiento y las características de los Active Objects para usarlos correctamente. Los fallos de implementación cuando se usan servicios asíncronos se traducen fácilmente en errores difícilmente detectables. A continuación se muestran algunas prácticas recomendadas para evitar estos problemas:

Es recomendable implementar un estricto control de errores, de forma que se pueda capturar cualquier tipo de error para saber el punto donde se ha producido.

No es recomendable que el método `RunL()` sea muy largo, ya que durante la ejecución de este método todos los Active Objects del thread quedan bloqueados. Si la ejecución de este método no es rápida se pueden producir efectos negativos para la percepción del usuario, como por ejemplo que el terminal no reaccione ante la interacción con el teclado durante un cierto tiempo.

Las peticiones asíncronas se pueden convertir en peticiones síncronas. Symbian OS ofrece la función estática `User::WaitForRequest(iStatus)`. Si se llama a esta función después de hacer una petición asíncrona y se le pasa la misma variable `iStatus`, el Active Object quedará bloqueado hasta que el servidor notifique que la petición ha sido procesada, obteniendo así el mismo comportamiento que usando llamadas síncronas. Esta práctica no es recomendable cuando se utilizan Active Objects, ya que si la petición no se llega a procesar el thread queda bloqueado. Esto supone que, si se estaban utilizando distintos Active Objects en el mismo thread, queden también bloqueados y se interrumpa la ejecución normal de la aplicación.

3.3. La API de sockets

Symbian OS ofrece una API de sockets para facilitar la adición de capacidades de comunicación a las aplicaciones. Esta API provee una serie de clases que permiten utilizar sockets, respetando la arquitectura cliente/servidor de Symbian. Estas clases se encargan de comunicarse con el servidor de sockets del dispositivo, ocultando así las capas inferiores de software que se utilizan para establecer la comunicación.

3.3.1. El servidor de sockets

El uso de la arquitectura cliente/servidor ofrece varias ventajas cuando se trabaja con sockets. El servidor de sockets se comunica con distintos software

de bajo nivel para cumplir con sus funciones. Hay que considerar, por ejemplo, las capas de transporte. Esto permite que los desarrolladores puedan utilizar las clases que proporciona la API de sockets, comunicándose únicamente con el servidor de sockets mediante programación de alto nivel.

Otra ventaja significativa es la posibilidad de usar módulos de protocolo. Muchos sistemas que implementan sockets están limitados a usarlos únicamente sobre redes TCP/IP. El servidor de sockets de Symbian OS va más allá, utilizando una arquitectura que soporta plug-ins (llamados módulos de protocolo). Esto permite a Symbian introducir nuevos protocolos y capas de transporte a medida que lo considere necesario. De esta forma Symbian OS se puede adaptar a cualquier entorno sin tener que hacer cambios en la arquitectura del servidor de sockets. Además permite a las aplicaciones extender fácilmente sus capacidades de comunicación a los nuevos entornos sin grandes modificaciones.

La API de sockets ofrece dos clases principales que son estrictamente necesarias para poder establecer una comunicación: `RSocketServ` y `RSocket`.

Además ofrece otras dos clases a destacar, orientadas a la resolución de nombres de host: `RHostResolver` y `RNetDatabase`.

A continuación se detallan las funciones de estas clases.

3.3.2. Clase RSocketServ

Esta clase representa una sesión de la aplicación cliente con el servidor de sockets.

La aplicación cliente no usará esta clase para enviar o recibir datos, ni para establecer la conexión entre los puntos finales. `RSocketServ` permite establecer la comunicación con el servidor de sockets de Symbian OS, de modo que cualquier aplicación que utilice sockets tendrá que tener una instancia a esta clase.

El método `Connect()` es el más importante de la clase `RSocketServ`. De hecho, no es necesario utilizar ningún otro método de esta clase para utilizar sockets convencionales (TCP o UDP).

Establecimiento de una conexión con el servidor de sockets

El método `Connect()` sirve para establecer la conexión con el servidor de sockets. El único parámetro que pasa es un entero que indica el número máximo de slots que se reservan en el servidor. Estos slots representan el número máximo de peticiones asíncronas que la aplicación puede solicitar al servidor de forma simultánea. Cada llamada a un método asíncrono consume

un slot, y éste quedará ocupado mientras la petición no haya sido procesada. Por otra parte, la utilización de métodos síncronos no consume ningún slot.

Una vez que el método ha retornado sin ningún error la aplicación esta preparada para abrir sockets, mediante instancias a la clase `RSocket`.

3.3.3 Clase RSocket

Cada instancia de esta clase representa un punto final de conexión (socket). Esta clase proporciona un gran número de métodos que permiten gestionar todos los aspectos de una conexión entre sockets, tales como:

- Servicios de gestión de la conexión, tanto para aplicaciones cliente como servidor.

- Establecimiento o resolución de la dirección local, así como resolución de la dirección destino.

- Lectura y escritura de datos en el socket.

Para poder utilizar estos métodos la clase `RSocket` necesita acceder al servidor de sockets de Symbian OS. Para ello es necesario que la aplicación disponga de una instancia a la clase `RSocketServ`, que actúa como intermediario entre la clase `RSocket` y el servidor de sockets. Como se ha comentado anteriormente, una sola instancia de `RSocketServ` puede gestionar múltiples sockets, de forma que se puede considerar a esta clase como un contenedor de objetos `RSocket`.

En la creación del socket se permite especificar, además del protocolo, el tipo de conexión a utilizar. Se puede clasificar este tipo de conexión en dos: orientado a conexión o no orientado a conexión. La clase `RSocket` ofrece métodos comunes y específicos para ambos casos. A continuación se muestran las diferentes fases, así como los métodos necesarios para establecer y utilizar una conexión entre sockets. Únicamente se muestran las fases correspondientes a sockets no orientados a conexión, ya que éstos han sido objeto de estudio durante la realización del proyecto.

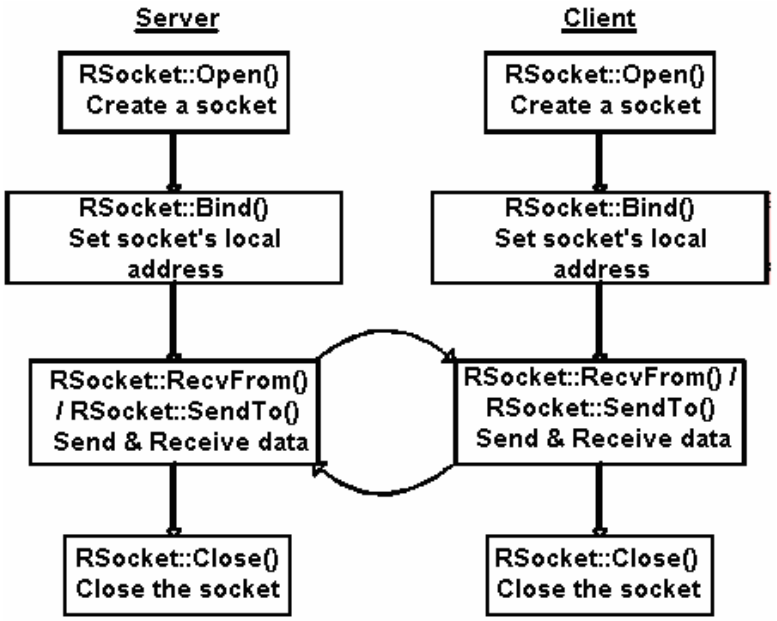

Fig. 3.2. Esquema de las fases de UDP.

1. Creación: Antes de poder utilizar un socket es necesaria su creación. Ésta se lleva a cabo a través del método `Open()`. Este método tiene distintas sobrecargas. El único parámetro que es común a todas ellas es una referencia a una instancia a la clase `RSocketServ`, ya que es en este momento donde se asocia al socket con la sesión del servidor de sockets de Symbian OS. Como paso previo, la sesión tiene que haber establecido la conexión con el servidor de sockets, tal y como se explicaba en el punto anterior.
   Adicionalmente se puede especificar la familia de direcciones (por ejemplo, la familia de direcciones de Internet), así como el tipo de socket (orientado a conexión o no) y el protocolo a utilizar.

2. Binding: Si la aplicación pretende utilizar un sockets para recibir datos es imprescindible asociar dicho socket con una dirección. Mediante el método `Bind()` se asocia el socket con un puerto local. De esta forma los datos que se reciban por este puerto podrán ser entregados al socket.

3. Envío y recepción de datos: Al usar sockets no orientados a conexión no son necesarias más fases para poder empezar el envío y recepción de datos, ya que no se establece conexión. Para ello la clase `RSocket` nos permite utilizar dos métodos: `SendTo()` y `RecvFrom()`. Mediante `SendTo()` podemos enviar un descriptor, especificando la dirección de destino, mientras que `RecvFrom()` se utiliza para recibir datos, pudiendo pasarle una referencia a un objeto dirección, obtener así la dirección del emisor de los datos.

4. Cierre del socket: Cuando se quiere terminar el uso de un socket que ha sido creado es necesario hacer una llamada al método `Close()`. De

esta forma nos aseguramos que todos los recursos asociados al socket quedan liberados.

Hay que destacar que algunos de los métodos que utiliza la clase `RSocket` son asíncronos, especialmente los de lectura y escritura. Esto es así debido a que la API de sockets está pensada para utilizarse en aplicaciones que hagan uso de Active Objects para poder realizar lecturas y escrituras de datos simultáneamente. Adicionalmente hay métodos equivalentes que utilizan llamadas síncronas al servidor de sockets, de forma que se pueden utilizar en una aplicación que no use Active Objects. No obstante esta práctica no es recomendable debido a aspectos de diseño comentados en el punto anterior (véase 3.2.2.). El uso de Active Objects en la implementación de sockets se discutirá más adelante.

CAPÍTULO 4. Pila RTP

En este capítulo se describe la implementación de una pila RTP para Symbian OS. Veremos la arquitectura que forma el conjunto de módulos que han sido utilizados. A continuación se detallará cada uno de los módulos, justificando su utilización y describiendo la API que exportan.

Este proyecto ha sido implementado en su totalidad en el lenguaje de programación C++, lo que implica un diseño orientado a objetos. Para describir los aspectos de diseño de los diferentes módulos se utilizará notación UML, ya que proporciona una metodología para representar sistemas orientados a objetos.

4.1. Arquitectura

Como se ha podido ver en el capítulo 2, RTP es un protocolo de aplicación. Por lo tanto la pila RTP se ha implementado como un módulo de la aplicación de prueba, detallada en el siguiente capítulo.

A la hora de crear la pila RTP se ha optado por portar una librería ya implementada. Debido a que RTP es un protocolo muy extendido existe una gran cantidad de implementaciones de código libre. Por supuesto, ninguna está preparada para poder ser utilizada en Symbian OS. Para la realización del proyecto se ha escogido una librería RTP ligera y sencilla, preparada para ser utilizada es sistemas basados en Unix, Windows y Solaris. Esta librería ha sido modificada, respetando la compatibilidad original, adaptándola para poder ser utilizada en Symbian OS, utilizando la arquitectura cliente servidor y el patrón de captura de eventos que caracterizan a Symbian OS.

La mayoría de las modificaciones se han hecho sobre el código de la librería. No obstante nos hemos encontrado con algunas funcionalidades que necesitaban ser implementadas como módulos independientes, dada su complejidad:

- Acceso a la capa de transporte: Se ha elegido UDP como protocolo de transporte. Los motivos están detallados en el capítulo 2, dado que este protocolo es el que mejor se adecua para ser utilizado por debajo de RTP. El problema que hemos encontrado es que la API de sockets de Symbian OS está diseñada para ser implementada con una arquitectura basada en Active Objects. Esto ha supuesto que se haya tenido que crear un módulo específico para el acceso a la capa de transporte.

- Listas: Symbian OS tiene una implementación de listas propia. Esto hace que sean incompatibles con las listas estándar, implementadas en la STL (Standard Template Library). La librería RTP que hemos seleccionado utiliza la STL, por lo tanto se ha tenido que crear un

módulo con las listas estándar para Symbian. De esta forma se pueden utilizar las listas de la librería sin tener que modificarlas.

El siguiente esquema muestra arquitectura que forman los diferentes módulos del proyecto. Además de los módulos de RTP aparecen la aplicación y la capa de transporte.

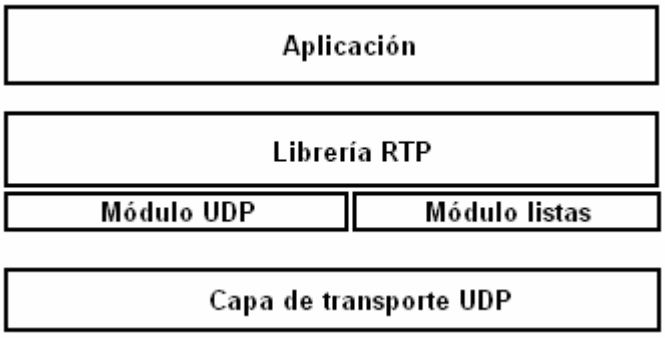

Fig. 4.1. Arquitectura del sistema.

4.2. Librería RTP

Como se ha mencionado anteriormente, para la realización del proyecto se ha optado por utilizar una librería RTP ya implementada. La librería elegida es la JRTPLIB, en su versión 3.4.0. La JRTPLIB (Jori's RTP Library) es el resultado de la tesis realizada por Jori Liesenborgs, sobre VoIP en entornos de red virtuales.

4.2.1. Motivación de su utilización

Realizar una pila RTP nativa para Symbian desde cero quedaba, por limitaciones temporales, fuera del alcance de este proyecto. Como alternativa se ha decidido utilizar la JRTPLIB, modificándola para que también pudiese funcionar en Symbian OS. Se ha elegido esta implementación por los siguientes motivos:

- En primer lugar, esta librería está implementada en C++, siguiendo un diseño orientado a objetos. Esto hace que la JRTPLIB pueda ser adaptada fácilmente a Symbian OS, pudiendo obtener una pila RTP en el lenguaje nativo de este sistema operativo.

- El hecho de que su diseño sea orientado a objetos también supone otra ventaja. La sencillez y claridad con la que está diseñada hace que el trabajo de adaptarla a Symbian OS pueda ser mucho menos costoso, ya que gracias a la modularidad con la que está implementada se pueden aislar fácilmente las diferentes secciones que deben ser modificadas. De hecho, la mayoría de las clases no

han tenido que ser modificadas, mientras que los grandes cambios se han centralizado en un limitado numero de clases.

- Otra ventaja la encontramos en que es una librería ligera, por lo que se adapta perfectamente a una plataforma móvil. No obstante, aún siendo sencilla y entendible, esta librería cumple con todos los requisitos necesarios para llevar a cabo comunicaciones en tiempo real a través de RTP.

- Otro de los aspectos que ha llevado a su elección es la generación de mensajes RTCP. Ésta está controlada internamente por la librería, de forma que el desarrollador de la aplicación no tiene que preocuparse de su creación.

Con estas características, la librería JRTPLIB se perfila como una buena elección para cumplir los objetivos del proyecto.

4.2.2. Funcionalidades

La librería permite seleccionar fácilmente el destino de los paquetes de datos, pudiendo elegir entre destinatarios unicast o un grupo multicast. En el caso de que muchos paquetes necesiten los mismos parámetros de envío, se pueden crear valores por defecto, de forma que se agiliza la implementación de aplicaciones.

A la hora de recibir los paquetes se soportan dos métodos. El primero está basado en el método de *polling*, el cual comprueba si hay datos RTP o RTCP entrantes. Este método no se adapta al modelo de Symbian OS, ya que se hace necesario un thread que se encargue de hacer las llamadas a esta función. Para adaptar la librería a Symbian se ha incluido un nuevo método para recibir los datos. Éste permite que la aplicación actúe como un observador, capaz de capturar un evento generado por la librería. Este evento se genera cada vez que llegan nuevos datos, lo que implica que los datos pueden ser procesados instantáneamente por la aplicación. De esta forma la JRTPLIB sigue el patrón de captura de eventos de Symbian OS.

La aplicación puede elegir entre tres modos de recepción. El primero acepta todos los paquetes entrantes. Con el segundo modo de recepción se aceptarán solo los paquetes de las fuentes que sean indicadas por la aplicación. Por último, existe un tercer modo en el que se aceptan todos los paquetes excepto aquellos cuyo origen haya sido denegado por la aplicación.

Los paquetes RTP y RTCP entrantes son procesados. La información que se extrae de ellos es guardada de forma independiente para cada usuario. La aplicación puede consultar estos datos recorriendo todas las fuentes, ya que éstas se guardan en listas, o seleccionando directamente una sesión RTP a través de su SSRC.

Como se ha comentado anteriormente, el protocolo RTCP esta totalmente implementado de forma interna. Cada vez que se envía o se recibe un paquete, la librería comprueba si es el momento de enviar un paquete RTCP. El intervalo entre envíos RTCP se calcula siguiendo la especificación de RTP publicada en el rfc 3550. Cuando se reciben paquetes RTCP se incluye la información relativa a la fuente en la lista de participantes de manera automática. De esta forma la aplicación puede consultar los parámetros estadísticos.

4.2.3. Arquitectura

Esta librería consta de un gran número de clases. A continuación se muestran las clases que son accesibles por la aplicación. Además se muestran las clases más importantes, que además han tenido que ser modificadas a la hora de adaptar la librería.

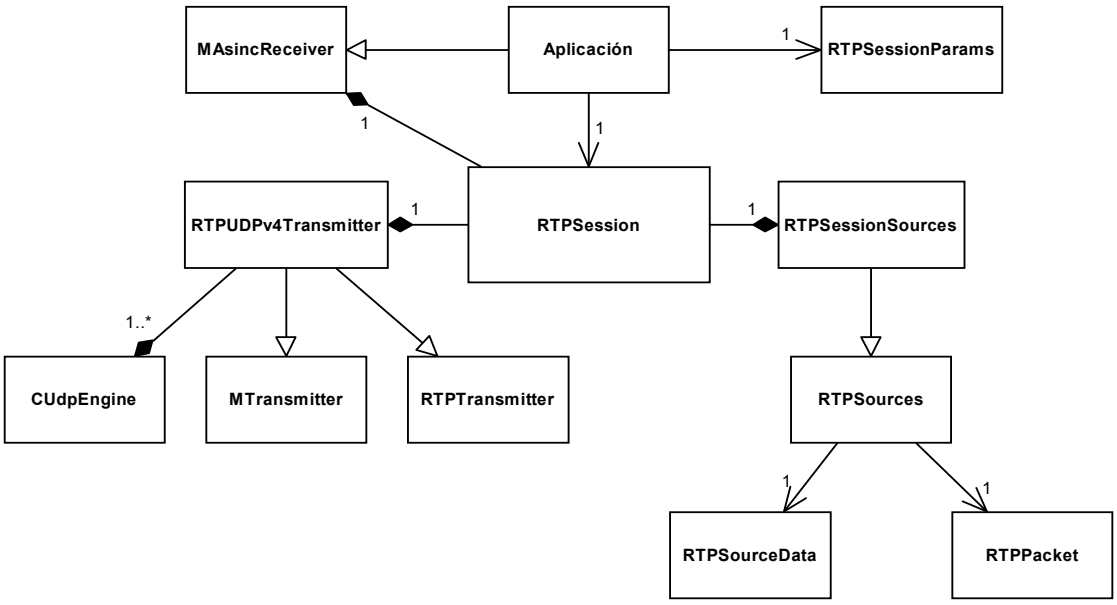

Fig. 4.2. Diagrama de clases de la JRTPLIB.

La API que exporta la librería consiste en cuatro partes:

- Clase `RTPSession`
- Clase `RTPSourceData`
- Clase `RTPPacket`
- Dos clases con parámetros.

`RTPSession` es la clase central de la librería. A través de ésta el usuario puede llevar a cabo todas las acciones propias de la librería: seleccionar los destinos, enviar paquetes RTP, comprobar si hay datos entrantes, etc. Esta clase además da al usuario la posibilidad de obtener información de cada

participante. Esta información se recopila a través de los mensajes RTCP y puede ser accedida por la aplicación mediante una instancia a la clase `RTPSourceData`. Para acceder a los datos de los paquetes RTP recibidos también se tiene que hacer uso de la clase `RTPSession`. Estos datos son pasados al usuario a través de instancias a la clase `RTPPacket`.

A continuación se describen los aspectos más relevantes de las clases que aparecen el diagrama, ordenadas de menor a mayor nivel de abstracción:

Clase RTPSession
Esta clase representa una sesión RTP. Como se ha comentado, es la clase más importante desde el punto de vista del usuario de la librería, ya que en muchas implementaciones es la única clase necesaria. `RTPSession` utiliza un gran número de clases para llevar a cabo todas sus funciones, de forma que actúa como enlazador de los distintos componentes de la librería. Los principales métodos vistos por la aplicación serán explicados en el capítulo 5, donde se describe la utilización de la librería.

Uno de los principales métodos internos de `RTPSession` es `ProcessPolledData`. Este método es llamado por la propia clase cada vez que se encuentran nuevos paquetes de datos entrantes. Es importante conocer la existencia de este método porqué es a través de éste donde se procesan los datos.

Clase MAsincReceiver
Hemos implementado esta clase abstracta, la cual permite que la llegada nuevos datos se notifique a la aplicación en forma de evento. Para ello la aplicación deberá heredar de ésta y crear un enlace con las capas de más bajo nivel de la librería, a través del método `AddAsincUser()` de la clase `RTPSession`. La aplicación tendrá que implementar un método virtual, el cual será llamado por la librería cada vez que haya nuevos datos entrantes. De esta forma se podrán procesar los datos de forma inmediata, sin tener que crear un hilo de ejecución que se encargue de comprobar la existencia de datos de forma periódica.

Clase RTPSessionParams
Ésta es otra de las clases utilizadas por la aplicación. Describe los parámetros que van a ser usados por la instancia a la clase `RTPSession`. Todas la funciones que ofrece son opcionales, excepto una. Es imprescindible configurar una unidad propia de timestamp para crear la sesión RTP. Para ello esta clase ofrece el método `SetOwnTimestampUnit()`. Una vez creada la instancia a la clase y configurado este parámetro, `RTPSessionParams` debe ser pasada en la creación de la sesión RTP.

Clase RTPSourceData
La clase `RTPSourceData` contiene toda la información referente a un participante de la sesión. Las instancias a estas clases son guardadas en tablas de hash, indexadas por el identificador SSRC. De esta forma se puede tener un acceso rápido a la información de cada fuente. La clase ofrece una

gran cantidad de métodos que permiten al usuario de la librería conocer cualquier parámetro estadístico de una fuente, recibidos a través de RTCP, así como métodos útiles para la aplicación generados por la librería.

Clase RTPSources

Representa una tabla en la cual se guarda información acerca de los datos de los participantes. Ofrece métodos que permiten procesar datos RTP y RTCP, así como iterar entre las distintas fuentes y obtener aquellas fuentes que contengan datos entrantes. Los métodos de `RTPSources` son usados por la clase `RTPSession` para llevar a cabo todas las acciones que tienen que ver con la obtención de datos y su procesado. `RTPSession` utiliza una clase que hereda de esta, extendiendo algunas funciones.

Clase RTPPacket

Esta clase representa un paquete RTP. La aplicación crea una instancia a esta clase y, mediante un método de `RTPSession`, se parsea un paquete RTP entrante y se genera una instancia de esta clase. Ofrece métodos para poder acceder a todos los campos del paquete RTP incluyendo, por supuesto, obtener el payload.

Clase RTPTransmitter

Se trata de una clase abstracta que especifica la interfaz para los componentes de transmisión. La librería cuenta con dos implementaciones: una para UDP sobre IP versión 4 y otra para UDP sobre IP versión 6. También existe un tercer componente de transmisión, pero no está implementado. Este componente, llamado `UserDefinedProto`, ha sido creado para que se puedan extender las capacidades de la librería, pudiendo así crear soporte para nuevos protocolos. No obstante, para la realización de este proyecto se deseaba utilizar UDP sobre IP en su versión 4, por lo que se ha decidido adaptar este componente existente en vez de crear uno nuevo para Symbian.

Por lo tanto, la clase que nos ocupa es `RTPUDPv4Transmitter`. Ésta hereda de `RTPTransmitter` e implementa la transmisión a través del protocolo UDP, sobre IP versión 4. Esta clase es una de las de más bajo nivel de la librería. Se encarga de acceder a la capa de transporte, de modo que ofrece métodos para enviar y recibir los paquetes RTP y RTCP. Para establecer las comunicaciones con otros participantes se utilizan sockets convencionales (para Unix y Windows). Para el caso de Symbian OS, hemos implementado una clase especial, llamada `CUdpEngine`. Esta clase representa algo más que un simple socket: contiene un conjunto de métodos que permiten al usuario del *engine* UDP (en este caso la librería) utilizar las funcionalidades de los sockets de Symbian, haciendo transparente los mecanismos utilizados en las comunicaciones de Symbian OS (véase 3.3.).

Clase MTransmitter

La definición de esta clase abstracta se encuentra dentro de la cabecera del *engine* UDP. Su función es proporcionar una vía de comunicación entre la clase `CUdpEngine` y `RTPUDPv4Transmitter`. Esta última tendrá que heredar de `MTransmitter`, teniendo que implementar un método virtual que podrá ser

llamado desde el *engine* UDP. A través de este evento el *engine* UDP puede notificar a la librería de la llegada de nuevos datos. En el próximo capítulo podremos ver la interacción entre las diferentes clases, que permite que el evento que informa de la llegada de nuevos datos se vaya propagando por los diferentes niveles de la libraría, hasta llegar a la aplicación.

4.2.4. Modificaciones mayores en la librería

Se han realizado muchas modificaciones en la librería para adaptar su uso a Symbian OS. No obstante, estas modificaciones se han visto concentradas en tres componentes de la librería.

4.2.4.1. Interacción con la capa de transporte

El sector que más cambios ha tenido que sufrir es, sin duda, la parte encargada de las comunicaciones con la capa de transporte, cuya representación es la clase `RTPUDPv4Transmitter`. La JRTPLIB está implementada para soportar sockets estándar, lo que le proporciona compatibilidad con un gran número de plataformas. No obstante, tal y como se pudo ver en el capítulo 3, Symbian OS no utiliza estos sockets. El hecho de que Symbian utilice un modelo específico para las comunicaciones mediante sockets, basado en multitarea no apropiativa, hace que las modificaciones necesarias para adaptar la librería a la nueva plataforma vayan más allá de incluir un nuevo tipo de sockets.

Introducir este nuevo tipo de sockets supone la inclusión de un conjunto de clases a la librería. Hacer estos cambios directamente en la clase `RTPUDPv4Transmitter` no es una buena práctica, debido al conjunto de módulos (Active Objects, clases diferentes para la lectura y escritura, etc.) que se tendría que incluir. Para preservar la estructura y las funciones de la clase `RTPUDPv4Transmitter` se ha optado por realizar un nivel más de abstracción, específico para Symbian OS y representado por la clase `CUdpEngine`.

La clase `CUdpEngine` representa un *engine* de sockets UDP para la plataforma Symbian. Ofrece un conjunto de métodos que permite utilizar una instancia a esta clase de forma similar a como se utilizan los sockets convencionales, ocultando a la librería el modelo de comunicaciones de Symbian OS. De esta forma se consigue añadir soporte a la nueva plataforma de forma clara y entendible, únicamente cambiando la utilización de sockets convencionales por instancias a la clase `CUdpEngine` en las líneas de código en que se hacen las llamadas a métodos de los sockets.

Como se ha mencionado anteriormente, se ha añadido un mecanismo de notificación de la llegada de datos entrantes por eventos. Esto supone que, cuando el *engine* haya recibido nuevos datos tendrá que avisar a la clase `RTPUDPv4Transmitter`. Para ello se ha creado una clase abstracta de la cual `RTPUDPv4Transmitter` tiene que heredar. Esta clase contiene un único

método virtual llamado `PollSymbian`. La implementación de este método parsea los datos recibidos en una instancia a la clase `RTPPacket` y llama al método `ProcessPolledData` de la clase `RTPSession`. De esta forma se procesa el paquete de forma inmediata.

4.2.4.2. Funcionalidades añadidas a la sesión RTP

El hecho de añadir un nuevo método de recepción nos ha obligado a añadir funcionalidades a la clase `RTPSession`. En primer lugar se ha incluido la definición de la interfaz `MAsincReceiver` en la cabecera de esta clase. De esta forma, cuando la aplicación hereda de esta interfaz, se le puede notificar de la llegada de nuevos datos entrantes.

Se ha añadido un método llamado `AddAsincUser`, que recibe una referencia a una instancia de la clase `MAsincReceiver`. Este método se utiliza desde la aplicación con el fin de informar a la sesión de que se desean recibir notificaciones por eventos. Si se ha activado el modo de recepción asíncrona, cada vez que se procese un paquete se llamará al método virtual `DataReceived` implementado en la aplicación (es el método de la interfaz `MAsincReceiver`).

Por último, se han añadido dos funciones que permiten interactuar con el *engine* UDP. El primer método es utilizado para obtener la dirección IP del dispositivo. La adición de este método se debe a que las funciones originales de la librería, que permiten obtener la IP local, no funcionan correctamente cuando se usan en Symbian OS. El segundo método, `StartReading()`, se utiliza para indicar al *engine* UDP que empiece a escuchar, a la espera de nuevos paquetes entrantes. Una vez llamada a esta función se pueden empezar a notificar eventos de llegadas a la aplicación, por lo que este método sirve para que la aplicación pueda realizar, si es necesario, acciones antes de empezar a procesar los paquetes entrantes.

4.2.4.3. Listas

Como se ha comentado anteriormente, se ha incluido una implementación de la STL para poder soportar las listas estándar en Symbian OS. La inclusión de este módulo ha sido sencilla: se ha incluido la referencia a la librería en todas las clases que utilizaban listas estándar. Esto ha sido suficiente para que Symbian OS utilice correctamente las clases de la STL utilizadas por la librería RTP.

4.3. Módulo UDP

La parte que se encarga de interactuar con la capa de transporte se ha implementado en un módulo a parte. Siguiendo la nomenclatura de Symbian,

este módulo recibe el nombre de *engine* UDP. Todas las funcionalidades que ofrece este módulo están representadas por la clase `CUdpEngine`.

4.3.1. Motivación de su utilización

Como se ha comentado anteriormente, Symbian OS no utiliza sockets estándar. Esto implica que se tengan que crear modificaciones en la librería para dar soporte a esta plataforma.

Symbian OS propone un modelo de comunicaciones basado en la arquitectura cliente servidor, utilizando multitarea no apropiativa. Por lo tanto, el añadir capacidades de comunicación para Symbian va más allá de incluir un nuevo tipo de sockets a la librería. Para poder implementar estas capacidades es necesario utilizar todos los elementos vistos en los capítulos 3.2 y 3.3. Los sockets de Symbian no se comunican directamente con la capa de transporte, sino que se tiene que crear una clase que representa una sesión con el servidor de sockets. Además se hace necesaria la utilización de Active Objects para el envío y la recepción de la información.

Todo esto supone que se necesiten varias clases para implementar las funcionalidades de los sockets estándar. La inclusión de esta arquitectura directamente en la librería hubiese supuesto una modificación poco comprensible y difícil de aislar a la hora de comprobar su funcionamiento. Por ello se ha decido crear un nivel más de abstracción, creando una clase con un conjunto de métodos, de forma que se represente todo el *engine* UDP. Esta clase ofrece un conjunto de métodos similares a los de los sockets estándar, de forma que se puede utilizar de forma similar, sin hacer grandes cambios en las clases originales de la librería.

El hecho de tener aislados estos componentes, representados por una única clase, nos ha dado la posibilidad de realizar aplicaciones de prueba exclusivas para el *engine* UDP. Teniendo en cuenta los problemas que han surgido durante su implementación, podemos asegurar que la utilización de un módulo independiente a la librería es la metodología más eficiente y clara.

4.3.2. Funcionalidades

La principal funcionalidad que ofrece el módulo de UDP es la comunicación con la capa de transporte de forma transparente, ocultando al usuario del modulo los aspectos relacionados con Symbian OS, como lo son el uso de Active Objects y la arquitectura cliente servidor.

El *engine* UDP permite enviar paquetes a través del protocolo UDP, sobre IP versión 4. En el momento del envío de los datos se permite especificar una dirección de destino, formada por la dirección IP y el puerto del destinatario.

Para poder recibir paquetes se ofrece la posibilidad de enlazar el módulo con un puerto local, de forma que todos los paquetes que lleguen a este puerto

puedan ser procesados. Como se ha explicado anteriormente, el método de recepción de datos funciona mediante notificaciones por eventos. Así el usuario del módulo UDP tendrá implementada una función virtual en su código, que será llamada por el módulo cada vez que se reciban datos entrantes.

El usuario puede especificar el momento en el que el *engine* empieza a escuchar por el puerto seleccionado. De esta forma se puede crear una instancia al *engine* en cualquier momento, sin tener que empezar a recibir datos a partir de ese instante. También cuenta con la posibilidad de detener el *engine* UDP una vez que no se necesite recibir o enviar más datos.

Por último, cuenta con un método que permite obtener la dirección IP local del dispositivo.

4.3.3. Arquitectura

A continuación se muestran todas las clases que intervienen en las comunicaciones con la capa de transporte UDP, así como las relacionadas con el uso de multitarea no apropiativa.

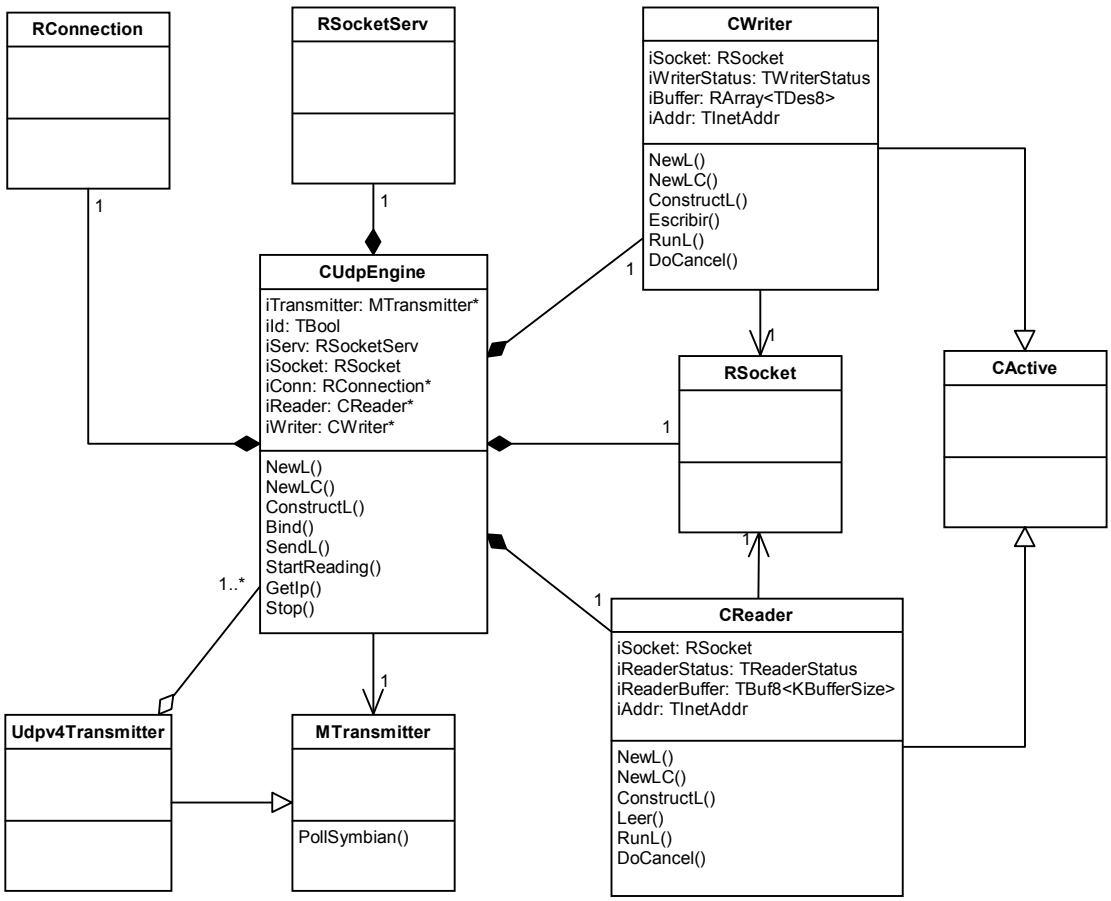

Fig. 4.3. Diagrama de clases del *engine* UDP.

La API que exporta este módulo consta únicamente de dos clases. La primera es la interfaz `MTransmitter`, la cual sirve únicamente para poder comunicar la llegada de datos entrantes al usuario del módulo UDP. La segunda es la clase `CUdpEngine`. Ésta es la representación del *engine*, ofreciendo métodos para interactuar con la capa de transporte de forma transparente.

El diseño de este módulo está basado en la utilización de dos clases independientes para la lectura y la escritura. Cada una de estas clases es un Active Object, de manera que se utiliza multitarea no apropiativa, realizando peticiones asíncronas al servidor de sockets de Symbian OS. Este modelo responde a una metodología marcada por Symbian, en la que se dan las facilidades necesarias para implementar multitarea a través de Active Objects en vez de usar threads independientes o bloquear la ejecución del módulo, a la espera de que se resuelvan las peticiones en el servidor de sockets.

A continuación se detallan los elementos que aparecen en la figura 4.3, excepto las clases RSocketServ, RSocket y CActive, que están explicadas en el capítulo 3.

Clase CUdpEngine
Como única representante del módulo, sus funcionalidades son las mismas que se han detallado en el apartado anterior. Esta clase actúa de enlace entre los distintos componentes. Los métodos que ofrece son los siguientes:

- Construcción: Siguiendo el modelo implantado por Symbian, la construcción de la clase se realiza mediante tres métodos: `NewL()`, `NewLC()` y `ConstructL()`. Esto se conoce como el constructor en dos fases, utilizando los dos primeros métodos para reservar el espacio de memoria en el terminal de forma segura y el tercero para llevar a cabo todas las acciones necesarias para inicializar la clase. Al crear la clase se recibe un puntero a una instancia de la clase `MTransmitter`. La aplicación, que hereda de esta interfaz, pasa un puntero a sí misma en la construcción del objeto. Esta referencia se utilizará para notificar a la aplicación de la llegada de nuevos datos entrantes. También recibe un segundo parámetro. La función de éste es poder indicar al usuario si los datos entrantes son del *engine* RTP o del RTCP. Por último, se crea la conexión con el servidor de sockets de Symbian. Esta conexión queda representada en la instancia de la clase `RSocketServ`. Esta conexión se abre y se activa mediante los métodos `Open()` y `Start()` de la clase `RConnection`.

- `Bind()`: La construcción del *engine* UDP no se completa hasta que se llama a este método, que recibe como parámetro una dirección. Esta dirección contiene el puerto local al que se desea enlazar el socket de escucha. Teniendo el valor de este puerto se enlaza una instancia de la clase RSocket con la instancia de la clase RSocketServ, especificando el tipo de protocolo a usar. A continuación se enlaza el socket con el puerto local. Una vez realizados estos pasos, el socket está listo para enviar y recibir

información. Por último se crean las instancias a las clases encargadas de la escritura y la lectura (CWriter y CReader), pasándoles la instancia al socket.

- `SendL()`**:** Este método sirve de enlace entre el usuario del módulo UDP y la clase `CWriter`. Se recibe un buffer con los datos a enviar y la dirección de destino. Estos dos parámetros son enviados al objeto encargado de la escritura, mediante la utilización del método `Escribir()` de ésta.

- `StartReading()`**:** Al igual que sucede con el método anterior, este hace de enlace entre el usuario y la clase `CReader`. Este método hace una llamada a la función `Leer()` del objeto encargado de la lectura para que empiece a escuchar por el puerto local a la espera de nuevos datos entrantes.

- `GetIP()`**:** Este método retorna un entero de 32 bits, el cual contiene la dirección IP del dispositivo.

- `Stop()`**:** Mediante este método se cierra el socket, terminando así la comunicación con el servidor de sockets de Symbian. También se detiene la ejecución de los Active Objects de lectura y escritura.

Clase MTransmitter

Esta clase abstracta contiene un único método virtual, llamado `pollSymbian()`. El usuario del *engine* UDP deberá heredar de esta clase e implementar este método, que recibe como parámetro un buffer con datos. Cuando llegan nuevos datos entrantes, el *engine* llama a este método, pasando el buffer con los datos a la aplicación.

Clase RConnection

Ésta es una clase perteneciente a la infraestructura de comunicaciones de Symbian OS. Su función es crear e inicializar una conexión con una interfaz de red. Se utiliza, junto con la instancia al servidor de sockets, para iniciar la comunicación entre las diferentes capas hardware y software que permiten la utilización de sockets. Además se puede utilizar para recibir notificaciones sobre el estado de la interfaz de red.

Clase CWriter

Esta clase hereda de `CActive`, por lo que es un Active Object. A través de esta clase se procesan todas las peticiones de escritura en el socket de forma asíncrona, ofreciendo así la posibilidad de que la ejecución de la aplicación continúe mediante multitarea no apropiativa.

No obstante, el uso de Active Objects para la escritura de datos en el socket presenta un problema difícil de detectar. Cuando se envía un paquete a través del socket se activa el Active Object para que, una vez procesada la petición asíncrona, el Active Scheduler pueda notificárselo a través del método `RunL()`. Por lo tanto, en el período de tiempo entre la activación la llamada del Active

Scheduler, la instancia de `CWriter` estará en estado activo. El problema surge cuando se intenta enviar paquetes de datos con una alta cadencia. Esto provoca que se intenten enviar varios paquetes mientras la instancia de CWriter se encuentra en estado activo, esperando a que se procese una petición de envío. Esto se traduce en la pérdida periódica de intervalos de paquetes.

Para solventar este problema se ha implementado un mecanismo de colas. Si se intentan enviar paquetes en el período que la clase está activa, se incluyen dichos paquetes en una lista. Esta lista será consultada en el método `RunL()`, ya que es en este método cuando la clase deja de estar activa. Si hay paquetes pendientes se enviarán sucesivamente hasta que la cola quede vacía. Si los paquetes llegan cuando la clase no está activa se enviarán normalmente.

La clase contiene los siguientes métodos:

- Construcción: Como todas las clases de Symbian, ésta tiene que implementar el constructor en dos fases, para lo que utiliza los métodos `NewL()`, `NewLC()` y `ConstructL()`. Este último método se encarga de añadir la instancia que acaba de ser creada al Active Scheduler. De esta forma el Active Scheduler incluirá a la instancia de la clase `CWriter` en su lista de Active Objects. El único parámetro que se recibe para la construcción es una referencia a un socket, que será utilizado para hacer las peticiones de escritura al servidor de sockets.

- `Escribir()`: Este método es llamado por `UDPEngine` cuando se quieren enviar datos. En primer lugar se comprueba si el Active Object está en estado activo. Si no lo está, se hace la petición asíncrona de escritura en el socket a través de la clase `RSocket` y, acto seguido, se avisa al Active Scheduler de que se está esperando la respuesta de una petición asíncrona, activando así el Active Object. En el caso de que la clase estuviese activa, se añaden los datos a una lista que actúa a modo de cola.

- `RunL()`: Este método virtual es llamado por el Active Scheduler cada vez que se confirma el envío de un paquete en el servidor de sockets. En este método se comprueba si hay paquetes pendientes de enviar. En caso de que así sea, se repite el proceso de envío descrito en el método anterior, eliminando de la cola los paquetes enviados hasta que ésta quede vacía. Cuando no hay más elementos en la cola termina la ejecución del método, dejando a la clase a la espera de nuevas peticiones de envío por parte de la librería.

Clase CReader

Esta clase también es un Active Object. Su función es recibir los datos entrantes, permitiendo que, durante la espera de dichos datos, se pueda continuar la ejecución de la aplicación sin tener un thread independiente para esta labor. Para ello se utiliza la multitarea no apropiativa de Symbian OS, heredando de la clase `CActive` y realizando peticiones asíncronas.

Siguiendo el modelo de Symbian OS, esta clase genera un evento cada vez que se reciben nuevos datos entrantes. La clase que está por encima de ésta (`RTPUDPv4Transmitter`) debe capturar este evento y recibir los datos, de forma que se puedan hacer llegar a la librería.

Para cumplir con estas funciones, `CReader` ofrece los siguientes métodos:

- Construcción: Para la creación de esta clase se utiliza el constructor en dos fases. Se reciben tres parámetros: En primer lugar una referencia a un socket, que será utilizado para recibir los datos. A continuación se recibe una referencia a la clase abstracta `MTransmitter`. Esta referencia representa a la clase `RTPUDPv4Transmitter`, que ha heredado de la interfaz. De esta forma se puede utilizar el método `PollSymbian()` para generar el evento. Por último se recibe un parámetro boleano, cuya función es la identificar al *engine* UDP que genera el evento. Esta variable es necesaria, debido a que la librería creará dos instancias del *engine*, una para RTP y otra para RTCP. Cuando uno de los dos genera un evento, la librería tiene que saber de cual de ellos se trata, para poder tratar la información como datos RTP o RTCP.

- `Leer()`: Este método es llamado por la clase `CUDPEngine` y por la propia clase `Creader`. Su función hacer una petición asíncrona al servidor de sockets de Symbian, a través la instancia a la clase `Rsocket`, y retornar después de haber activado el Active Object. Una vez hecha esta petición la aplicación puede continuar con su ejecución, sin tener que esperar a que lleguen datos entrantes. La llamada desde `CUDPEngine` solo se lleva a cabo una vez, para indicar al *engine* que debe empezar a recibir.

- `RunL()`: Cuando se reciben datos en el servidor de sockets, el Active Scheduler llama a este método. En su ejecución se genera el evento que informa a `RTPUDPv4Transmitter` de la llegada de nuevos datos, utilizando el método virtual `PollSymbian()`. Con este método se envían los datos, la dirección de origen de estos y el identificador del *engine*. Con estos tres parámetros la librería puede procesar el paquete correctamente.

4.4. Módulo de listas

Symbian OS ha optado por utilizar un sistema de listas y descriptores propio. Esto implica que no tiene implementada la STL (Standard Template Library). Esta librería es ampliamente aceptada como un estándar para la gestión de listas y descriptores.

La JRTPLIB utiliza la librería STL para guardar una gran cantidad de objetos en distintas listas. Esto supone una incompatibilidad a la hora utilizarla sobre

Symbian OS. Para solucionar este problema se ha implementado un módulo de adaptación.

4.4.1. Motivación de su utilización

A la hora de proporcionar compatibilidad en el uso de listas entre la librería y la nueva plataforma, hemos estimado dos opciones. La primera opción consiste en cambiar el tipo de listas de la libraría, substituyendo las que ofrece la STL por las propias de Symbian OS. Como segunda opción tenemos el caso contrario: crear un módulo de adaptación, con el fin de implementar las listas de la STL para que puedan ser utilizadas en Symbian.

Para la realización de este proyecto hemos optado por la segunda opción. Han sido, principalmente, dos factores los que nos han llevado a tomar esta decisión:

- La JRTPLIB utiliza un gran número de listas para varias funcionalidades. Esto implica que el trabajo de adaptar la librería al uso de las listas de Symbian sea muy costoso en tiempo y esfuerzo.

- En segundo lugar, existen una gran cantidad de implementaciones libres de la STL, diseñadas para varias plataformas. De esta forma no es difícil encontrar una implementación que pueda ser fácilmente adaptada a nuestras necesidades.

Para la implementación de la STL se ha escogido una libraría existente, diseñada para ser portada a dispositivos empotrados, llamada miniSTL. El hecho de que su diseño esté orientado ha dispositivos con grandes restricciones hace que se adapte perfectamente a las necesidades de los dispositivos móviles.

La implementación de un módulo independiente, que permita a Symbian utilizar las clases de la STL, nos ha permitido escapar de un sinfín de modificaciones en la librería RTP. Esto permite preservar la compatibilidad con las otras plataformas, así como respetar la implementación original de la JRTPLIB, sin riesgo a que algunas funcionalidades se viesen afectadas por las limitaciones de las listas de Symbian OS.

4.5. Secuencia de envío y recepción

Una vez vistas todas las capas que forman la pila RTP, así como las clases que las componen, podemos representar a modo de síntesis sus dos funcionalidades más importantes: el envío y la recepción de datos.

Con el fin de ver como interactúan las distintas clases entre sí, representaremos estas dos funcionalidades mediante diagramas de secuencia. A través de estos diagramas se espera sintetizar las funciones más importantes y los componentes que permiten el envío y la recepción de datos.

4.5.1. Secuencia de envío

A continuación se muestra el diagrama de secuencia de envío. En él se muestra desde que la aplicación llama al método `SendPaquet()` de la clase `RTPSession`, hasta que se confirma el envío del paquete en el servidor de sockets.

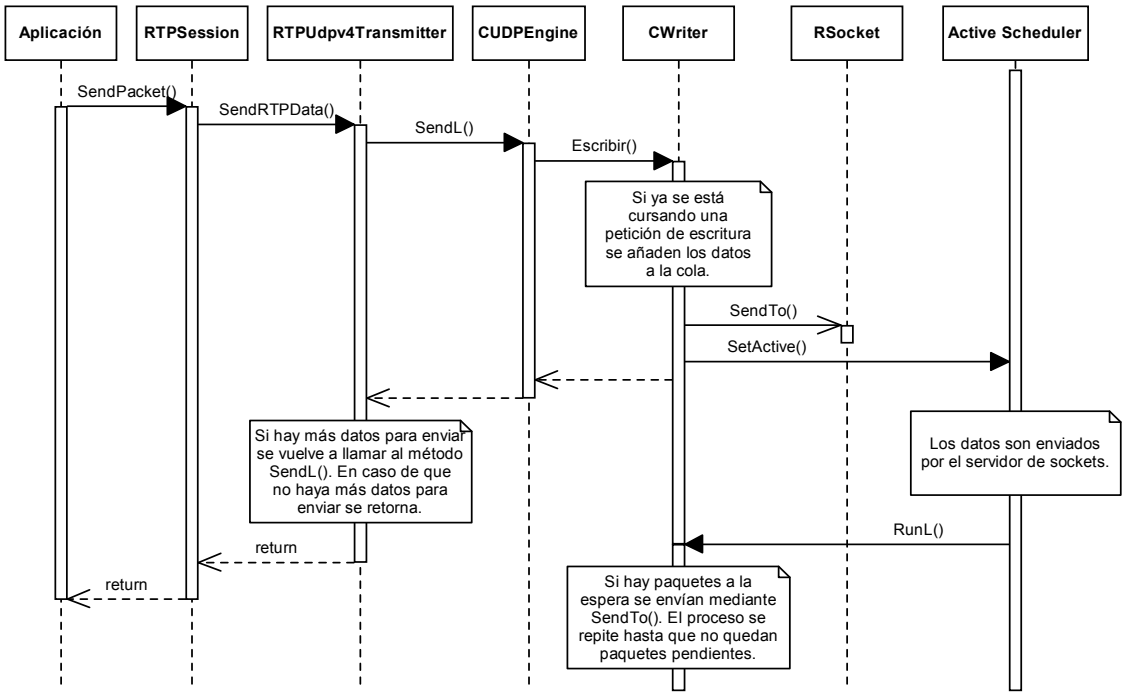

Fig. 4.4. Diagrama de secuencia de envío

Se puede observar como la petición de envío de datos se va propagando por los distintos niveles de la librería, hasta llegar a la clase `CWriter`. Esta clase comprueba si se están enviando datos. En caso de que así sea, se añade el paquete a una cola de espera. En caso contrario se envían los datos a través del socket, mediante el método asíncrono `SendTo()` y se retorna. Cuando el servidor de sockets de Symbian envía los datos el Active Scheduler llama al método `RunL()` de `CWriter`. En este método se comprueba si hay paquetes pendientes de enviar en la cola de espera, repitiendo el proceso de envío hasta que no quedan paquetes pendientes.

4.5.2. Secuencia de recepción

El siguiente diagrama muestra la secuencia de recepción de datos. A través de este diagrama se puede ver de forma más clara la interacción entre las clases que generan los eventos y las que actúan como capturadores de éstos.

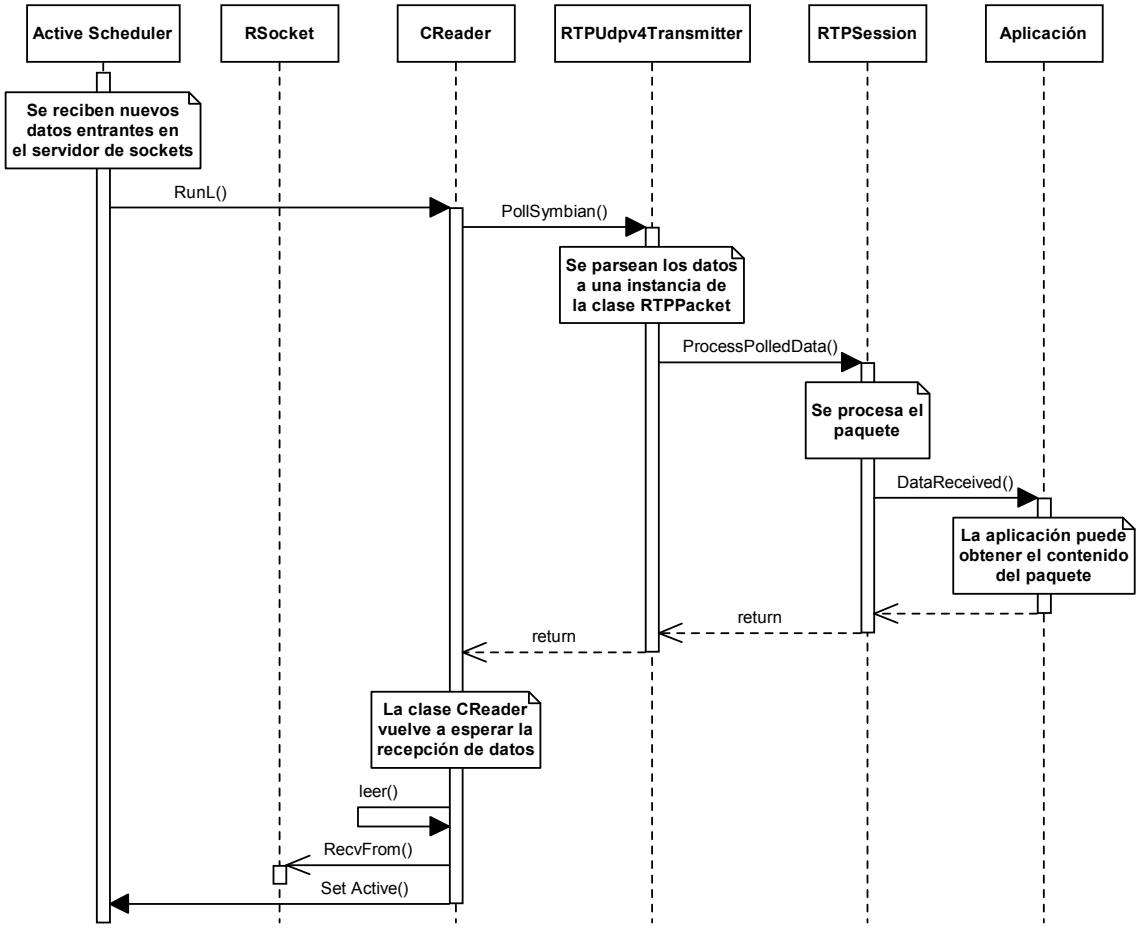

Fig 4.5. Diagrama de secuencia de recepción.

Al inicio de esta secuencia el servidor de sockets de Symbian OS se encuentra a la espera de nuevos datos entrantes. Cuando éstos llegan el Active Scheduler llama al método `RunL()` de la clase `CReader`. Esta clase genera un evento, llamando al método virtual `PollSymbian()`, que ha sido implementado por la clase `RTPUDPv4Transmitter`, actuando así como capturador de este evento. Una vez que la instancia de `RTPTransmitter` tiene los nuevos datos entrantes crea un objeto de la clase `RTPPacket` y le parsea los datos recibidos. Una vez creado el paquete, se le pasa a `RTPSession` para que lo procese. Al finalizar el procesado y haber colocado el paquete en la lista correspondiente, se avisa a la aplicación de la existencia de nuevos datos. Esto se hace mediante otro evento, representado en el método virtual `DataReceived()`, que tiene que ser implementado en la aplicación. De esta forma se completa la propagación del evento y la aplicación puede obtener los datos entrantes.

Los métodos de las clases retornan, hasta llegar de nuevo a `CReader`. Una vez llegados a este punto se llama al método `leer()`, que hace una petición asíncrona a la clase `RSocket` para que escuche a la espera de nuevos datos entrantes, completando el ciclo. De esta forma se consigue que la librería esté permanentemente a la espera de datos.

CAPÍTULO 5. Balances y conclusiones

5.1. Objetivos alcanzados

Los objetivos especificados al inicio del proyecto se han cumplido satisfactoriamente.

El propósito principal consistía en probar las posibilidades de implementar una pila RTP para Symbian OS. Durante el desarrollo del proyecto se ha conseguido migrar una librería existente, haciendo que ésta sea operativa sobre el sistema operativo Symbian OS. No se ha podido probar que su funcionalidad sea completa al ejecutarse sobre un terminal, pero mediante pruebas realizadas con emuladores, hemos podido comprobar que las principales funcionalidades, tales como enviar y recibir paquetes RTP, funcionan correctamente.

Se ha realizado un estudio teórico tanto del protocolo como de la plataforma sobre la que se ha implementado. El resultado ha sido muy satisfactorio, obteniendo conocimientos sobre estos dos aspectos que han sido aplicados directamente a la realización del proyecto, ayudando a comprender los inevitables problemas que han surgido, agilizando así la consecución del proyecto.

Por último, hemos diseñado una aplicación para Symbian OS con el fin de poder probar la funcionalidad que ofrece la librería migrada. Adicionalmente, se han incluido funcionalidades multimedia, lo que nos ha permitido entender más claramente la arquitectura del sistema, ya que hemos necesitado el acceso al hardware del dispositivo. Así mismo, la realización de esta aplicación nos ha servido para adentrarnos en la creación de interfaces gráficas

5.2. Impacto ambiental del proyecto

Al tratarse del desarrollo de software, el único impacto ambiental que produce viene dado por el gasto energético que se ha llevado a cabo durante el desarrollo del proyecto.

5.3. Posibles mejoras

Tal y como se ha comentado, no se ha podido probar el funcionamiento de la librería en un terminal, realizando las pruebas únicamente al emulador. Esto puede suponer la parición de problemas no vistos en el emulador. Asimismo, no se tiene constancia de la integridad de los datos RTCP generados por la librería. De forma que, una posible mejora sería la de crear un entorno de prueba, en el que se pudiese crear un escenario complejo, para así poder evaluar más fielmente las prestaciones de la librería RTP.

Otro aspecto que admite modificaciones es el módulo de streaming de audio. El objetivo de su implementación era meramente formativo. El resultado es la consecución del acceso al hardware del dispositivo, no obstante no se ha implementado ningún mecanismo de buffering, por lo que la reproducción del audio resulta discontinua. Una posible mejora podría consistir en incluir dicho mecanismo, de forma que se pudiese obtener una reproducción fluida.

5.4. Ampliaciones futuras

La implementación del protocolo RTP constituye el primer paso para la consecución de aplicaciones multimedia en tiempo real para dispositivos basados en Symbian. Una de las aplicaciones más interesantes de este tipo de comunicaciones es la telefonía IP. Una posible ampliación podría ser implementar una capa de señalización y control de sesión basada en SIP, de forma que se pudiese utilizar conjuntamente con la pila RTP. Esto permitiría la implementación de una aplicación de telefonía IP completamente funcional para Symbian OS.

BIBLIOGRAFÍA

[1] Harrison, R., *Symbian OS c++ for Mobile Phones*, Ed Wiley, Chichester, 2003.

[2] C++ Language Tutorial. Página web, URL
< http://www.cplusplus.com/doc/tutorial/>

[3] Symbian OS – the mobile operating system. Página web, URL
< http://www.symbian.com/>

[4] NewLC. Página web, URL
< http://newlc.com/>

[5] Forum Nokia – Developer resources. Página web, URL
< http://www.forum.nokia.com/main.html>

[6] RTP: About RTP and the Audio-Video Transport Working Group. Página web, URL
< http://www.cs.columbia.edu/~hgs/rtp/>

[7] rfc 3550 - RTP: A Transport Protocol for Real-Time Applications. En línia, URL
< http://www.ietf.org/rfc/rfc3550.txt>

[8] JRTPLIB Jori's page. Página web, URL
< http://research.edm.luc.ac.be/jori/page.html>

[9] Practical UML. Página web, URL
< http://bdn.borland.com/article/0,1410,31863,00.html>

ANEXO 1. Streaming de audio

Un stream de audio es un flujo continuo de datos, en el cual no está definido el principio y el final. Al contrario de lo que ocurre al reproducir archivos, al hacer streaming se reproduce el audio a medida que se va obteniendo de la fuente. De esta forma no es necesario, por ejemplo, esperar a que un archivo se descargue en su totalidad para empezar a reproducirlo. La fuente fragmenta el audio en pequeñas porciones que son enviadas al destinatario, el cual va reproduciendo estos fragmentos a medida que los va recibiendo. De esta forma se puede conseguir una gran reducción del retardo en la reproducción. La fuente debe codificar el audio con una baja tasa de bit, de forma que el destinatario pueda reproducir el audio de forma continua.

La aplicación de esta técnica en las comunicaciones de audio en tiempo real (como VoIP) es absolutamente imprescindible. El retardo en este caso debe ser muy pequeño, ya que éste es un servicio en el que los interlocutores tienen que poder interactuar unos con otros en tiempo real. Si la voz se retarda demasiado los usuarios pueden percibir el desfase y la experiencia de usuario se degrada.

A continuación veremos las posibilidades que Symbian OS ofrece a los desarrolladores para la gestión de contenidos multimedia, centrándonos en el streaming de audio.

A1.1. MMF (Multi Media Framework)

Para abarcar todas las posibilidades multimedia de los terminales, Symbian OS cuenta con el MMF (Multi Media Framework). Éste proporciona una API que provee a los desarrolladores de una serie de clases que permiten explotar las siguientes capacidades de audio y vídeo de los dispositivos:

- Reproducción, grabación y conversión de audio: Tres clases mediante las cuales se puede crear, reproducir y manipular datos de audio en ficheros, descriptores e incluso URLs.

- Reproducción y grabación de vídeo: Dos clases que permiten la creación y reproducción de video guardado tanto en ficheros como en descriptores o URLs. A diferencia con el audio, no se pueden hacer conversiones en la codificación del video mediante estas clases.

- Reproducción de tonos: Una única clase que proporciona métodos que permiten la reproducción de tonos.

- Streaming de audio: Además de la manipulación de audio convencional (a través de ficheros), la MMF ofrece dos clases que permiten reproducir y crear streams de audio.

A continuación se muestra cómo interactúan los distintos componentes de la MMF entre sí:

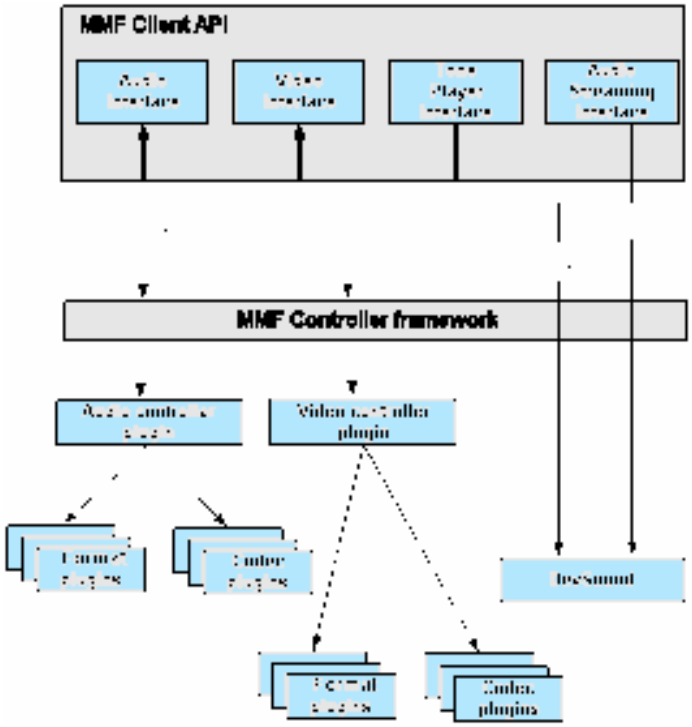

Fig. A1.1. Arquitectura de la MMF.

Como se ha comentado anteriormente, la MMF ofrece cuatro interfaces a través de la MMF Client API. Las interfaces de grabación y reproducción de audio y video interactúan con al MMF Controller framework. Esta capa tiene acceso a los plugins de audio y video, y éstos a su vez contienen los plugins con los formatos y los codecs. Al utilizar esta capa intermedia se permite reproducir, grabar y modificar audio y video mediante programación de alto nivel, haciendo transparente la codificación de los datos al programador.

Se puede observar que la interfaz de reproducción de tonos no necesita interactuar con la MMF Controller framework. Esto se debe a que los tonos son un formato de audio simple y no necesitan ninguna conversión a otros formatos.

Por último vemos que la interfaz de streaming de audio tampoco utiliza la capa de control de la MMF. Mediante streaming se podrían reproducir flujos de audio codificados en cualquier formato, con lo que parece necesaria la existencia de interacción con la MMF Controller framework, pero no es así. Los tipos de codificación soportados para la reproducción o grabación mediante streaming se ven limitados a PCM16 (a partir de la versión 8.0 se soportan algunos formatos más). De esta forma es posible que la interfaz pueda interactuar directamente con el servidor que se encarga de gestionar el dispositivo de sonido del terminal. Gracias a esto se consigue una rápida respuesta, esencial

en las comunicaciones en tiempo real, a cambio de una limitación en los formatos soportados.

Para la realización de este proyecto únicamente hemos necesitado la interfaz de streaming de audio. Por ello a continuación se mostrará el funcionamiento y las posibilidades que ofrece dicha interfaz.

A1.2. Streaming de audio en Symbian OS

La "Audio Streaming API" es la interfaz que Symbian OS ofrece para enviar o recibir datos de audio mediante streaming. Esta API proporciona dos clases: `CMdaAudioOutputStream` para el envío de streams y `CMdaAudioInputStream` para la recepción. Hay que recordar que estas clases interactúan directamente con el dispositivo de sonido de la MMF, permitiendo así una baja latencia en el procesado del audio. A continuación se muestra un esquema más detallado de los elementos que forman la API de streaming de audio:

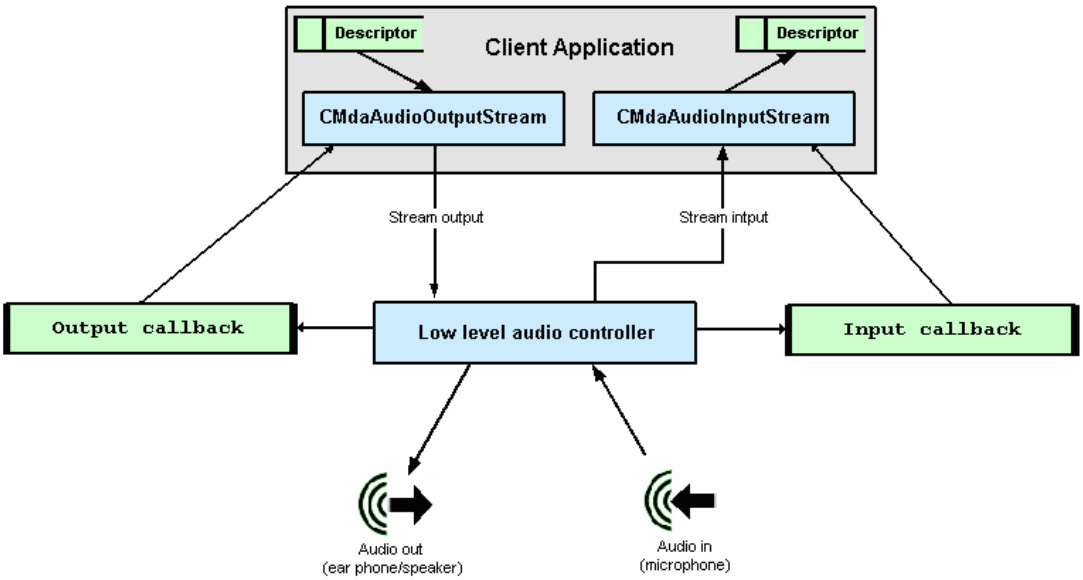

Fig. A1.2. Arquitectura de la Audio Streaming API.

En la figura se puede observar que cada una de las clases tiene una clase de tipo "callback" asociada. Gracias a estas clases se permite que la MMF pueda informar a la aplicación cliente de sucesos de forma asíncrona. Este mecanismo de retrollamada sigue el patrón de Symbian, haciendo que las aplicaciones se comporten como capturadores de eventos y permitiendo que se lleven a cabo diferentes tareas mientras se reproduce o crea el stream de audio.

Las clases "callback" actúan como observadores de eventos. Bajo este patrón, una aplicación que pretenda implementar streaming de audio deberá crear un

objeto que derive de la clase observadora. Estas clases son de tipo M, lo que, según la nomenclatura de Symbian, significa que son un conjunto de métodos puramente virtuales. En el caso de estos observadores, el objeto que derive tendrá que implementar tres métodos.

Como se puede observar, cada clase tiene un observador diferente, de forma que si una aplicación pretende utilizar stream de entrada y de salida deberá crear un objeto que derive de ambos observadores. Hay que decir que ambos observadores tienen los mismos métodos, únicamente variando el nombre para que se pueda distinguir que stream está generando el evento. En la práctica es común que la propia aplicación sea la que derive de los observadores, de forma que sea la aplicación la que tenga que implementar las funciones virtuales.

A continuación se muestran las diferentes fases y los métodos principales utilizados por las dos clases que forman la API de streaming de audio.

A1.2.1. CMdaAudioOutputStream

El stream de salida está constituido por dos clases: la instancia al stream (`CMdaAudioOutputStream`) y su respectivo "callback" (`MMdaAudioOutputStreamCallback`). El uso de esta interfaz conlleva las siguientes fases:

1. Creación: La instancia al stream de salida se puede crear con una llamada al método `NewL()`, pasando como referencia la clase que actúa de observador (callback).

2. Apertura: Es necesario abrir el stream llamando al método `Open()`. Previamente a este paso deberá haber sido creado un objeto de tipo `TMdaAudioDataSettings`, que será pasado al método `Open()`. Este objeto contendrá todos los parámetros del stream, como el número de canales o la tasa de muestreo. Esta llamada inicializa el stream y llama al método `MaoscOpenComplete()` del observador con un código de error. Si el proceso de inicialización se ha efectuado correctamente el código de error valdrá `KErrorNone`. Al igual que ocurrirá en los demás métodos virtuales del observador, dependerá del desarrollador el correcto tratamiento de los posibles errores.

3. Preparado: Una vez que la apertura se ha hecho correctamente, se pasa al estado preparado. Es posible modificar algunos parámetros del stream después de la llamada al método `Open()`, como por ejemplo el volumen utilizando el método `SetAudioPropertiesL()`. No obstante hay algunos parámetros que son invariables desde el momento de la apertura del stream, como la tasa de muestreo. En esta fase el stream está preparado para enviar buffers de audio al dispositivo de sonido de las capas más bajas de la MMF. Estos buffers deberán ser guardados en descriptores de 8 bits, siendo codificados en PCM16. Estos envíos se

realizan mediante una llamada al método `WriteL()`, pasando el buffer que se desea reproducir. La aplicación será notificada cada vez que un buffer haya sido copiado a través el método `MaoscBufferCopied()` del observador, pasando esta vez un puntero al buffer copiado además de un código de error. Al tener un puntero a buffer que ya ha sido copiado, éste se puede eliminar fácilmente.

4. Parado: Cuando la reproducción de audio se detiene se notifica a la aplicación cliente a través de método `MaoscPlayComplete()` del observador, pasando un código de error. Hay dos formas de detener un stream. La primera es llamando al método `Stop`(). De esta forma se obliga a las capas inferiores de la MMF a que dejen de reproducir buffers, incluso los que tenga pendientes. También se puede forzar la parada del stream si no se envían buffers hasta que el dispositivo de sonido haya consumido todos los buffers pendientes de reproducir. Al hacer esto el código de error resultante es `KErrorUnderflow`. Es bastante común que se obtenga esta situación sin ser forzada por el desarrollador. En este caso, si la aplicación cliente tiene más buffers para reproducir, significa que no se está entregando datos a la velocidad necesaria.

Una práctica común cuando se trabaja con streams de audio es el buffering. Los buffers de audio se pueden reproducir a medida que van llegando de la fuente, pero de esta forma normalmente se obtiene una reproducción discontinua y aparecen notificaciones de parada del stream con el código de error `KErrorUnderflow`. Una solución para evitar esto es hacer buffering, creando una pequeña cola con buffers de audio y enviándolos a las capas inferiores de la MMF a la vez. De esta forma se consigue que el dispositivo de audio siempre se encuentre ocupado y tenga siempre buffers pendientes de reproducir.

A1.2.2. CMdaAudioInputStream

Al igual que el stream de salida, el de entrada está formado por una instancia al stream y su respectivo “callback”. La utilización de éste es prácticamente idéntica al anterior, únicamente cambiando los nombres de los métodos y, por supuesto, la dirección del stream. En este caso grabará el sonido del micrófono del dispositivo. Cada vez que se haya grabado una porción de audio, el “callback” lo notificará a la clase `CMdaAudioInputStream`, enviándole un buffer con los datos. De esta forma se genera un flujo de datos de salida.

A1.3. Implementación de un módulo de streaming de audio

Utilizando la API de streaming que proporciona la Multi Media Framework, se ha implementado un módulo de streaming de audio.

La finalidad de éste es proporcionar a la aplicación la posibilidad de utilizar tráfico multimedia a la hora de probar las prestaciones de la librería RTP.

A1.3.1. Arquitectura del módulo de streaming

A continuación se muestran las clases que componen el módulo de streaming de audio.

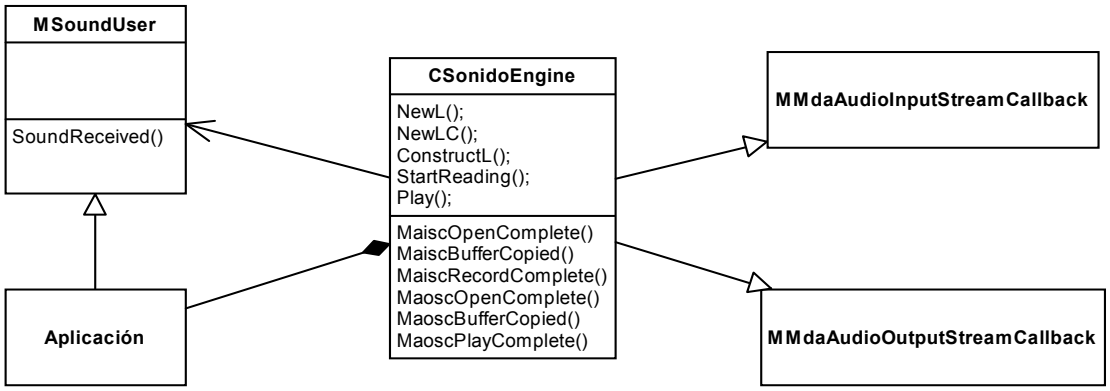

Fig A1.3.

La clase `CSonidoEngine` representa el módulo de streaming. Como se puede observar en el diagrama, esta clase hereda de los dos “callbacks”. Esto se debe a que el módulo ofrece la posibilidad de crear streams de audio en los dos sentidos. Por lo tanto, `CSonidoEngine` debe implementar los seis métodos virtuales que definen los dos “callbacks”.

La API que exporta el módulo es muy simple. Consta únicamente de dos métodos: uno para reproducir datos por el altavoz del dispositivo (`Play()`) y otro para que se empiece a generar un stream de sonido (`StartReading()`). A medida que el módulo genera paquetes de audio, lo notifica a la aplicación mediante eventos, utilizando el método virtual `SoundReceived()`, que debe ser implementado en la aplicación al heredar ésta de la interfaz `MSoundUser`.

En el siguiente anexo se podrá ver con más detalle la metodología para usar este módulo.

ANEXO 2. Aplicación de prueba

Una vez realizadas las modificaciones en la librería RTP, se ha procedido a la implementación de una aplicación. El objetivo de ésta es probar el funcionamiento final de la migración.

La realización de la aplicación nos ha permitido, además, introducirnos en la implementación de interfaces gráficas de Symbian OS. Adicionalmente, se ha incluido en la aplicación un módulo que permite la grabación y reproducción de audio por streaming, cuya descripción hemos podido ver en el anexo 1. La inclusión de este módulo nos ha dado la posibilidad de adentrarnos en las funciones multimedia de Symbian OS, así como crear un escenario más interesante para probar las capacidades de la pila RTP.

En este capítulo se detalla la implementación de la aplicación y se describen las clases que permiten la creación de interfaces gráficas en Symbian OS.

A2.1. Funcionalidades de la aplicación

El propósito principal de esta aplicación es realizar pruebas con la librería RTP. Para ello ofrece las siguientes funcionalidades:

A2.1.1. Notificaciones de estado

La interfaz gráfica consta de tres etiquetes de texto en las que se muestra el estado en el que se encuentra la pila RTP, indicando al usuario las acciones necesarias para poder comenzar a utilizar la sesión RTP. Asimismo, las otras dos etiquetes tienen la finalidad de informar sobre el estado de los datos enviados y recibidos, utilizándose para mostrar el contenido de los paquetes o el número de paquetes enviados y recibidos.

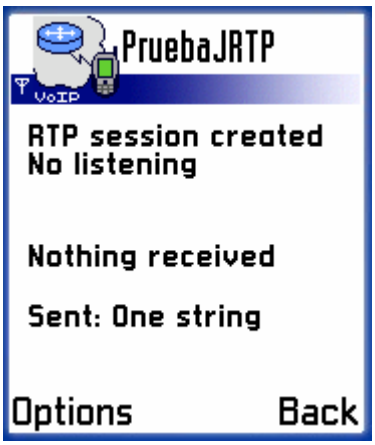

Fig. A2.1. Interfaz gráfica.

Asimismo, se utilizan cuadros de dialogo para informar al usuario de posibles errores, tales como intentar enviar un paquete o consultar la dirección IP, sin haber creado previamente la sesión RTP. Este control de errores se lleva a cabo en la aplicación. No obstante, la librería RTP también tiene mecanismos para resolver este tipo de errores.

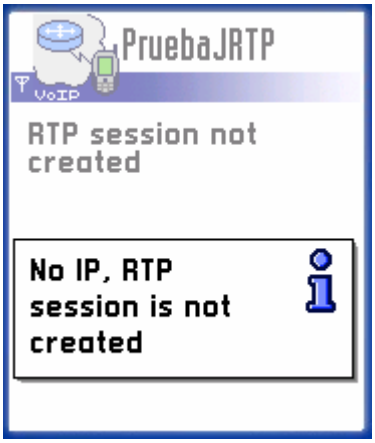

Fig. A2.2. Dialogo de aviso.

A2.1.2. Configuración de la sesión

La aplicación permite la creación de una sesión RTP. Para ello se debe establecer el puerto local, en el cual se esperará recibir los datos entrantes. En el momento de crear la sesión también se establece la conexión de red, apareciendo un diálogo propio de Symbian, en el que se permite seleccionar la conexión que se desea utilizar. Una vez creada, se informa a través de la etiqueta de la interfaz gráfica de que la sesión está creada, indicando al usuario que no se han establecido destinatarios, además de indicar que no se está escuchando para recibir datos entrantes.

En el mismo submenú de configuración, se permite añadir un destinatario. Se debe rellenar un formulario, compuesto por una dirección IP y un puerto de destino.

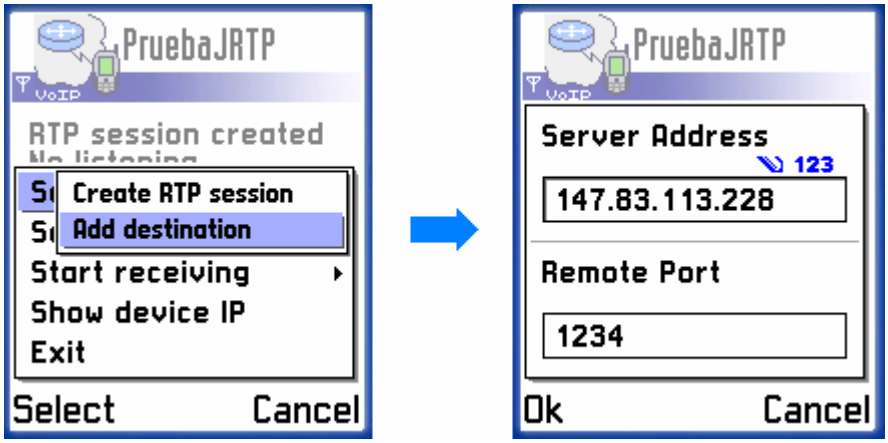

Fig. A2.3. Cuestionario para añadir destinatario.

A2.1.3. Envío de datos

Con la finalidad de probar las posibilidades de envío de datos de la librería, se han implementado tres tipos de envío.

El primero envía un único paquete, el cual contiene una cadena de texto. Esta función permite probar la conectividad entre dos participantes de una sesión RTP.

En segundo lugar, se ha implementado una función que envía diez paquetes. La finalidad de esta función es comprobar el correcto envío y recepción de una serie de paquetes consecutivos.

Por último, se ofrece la posibilidad de enviar paquetes de voz a un destinatario. Al utilizar esta función la aplicación hace uso del módulo de streaming de audio, capturando la voz a través del micrófono del dispositivo y generando el stream. De esta forma se envían paquetes con de forma continua, con una elevada cadencia.

A2.1.4. Recepción de datos

Cuando la aplicación está lista para empezar a recibir datos, utiliza la función de empezar a recibir. El usuario puede escoger entre recibir cadenas de texto o voz. La diferencia radica en la forma de presentar estos datos al usuario. Cuando se selecciona la recepción de cadenas, el payload de los paquetes recibidos se muestra en la etiqueta de recepción. En caso de haber seleccionado la recepción de voz, los datos recibidos son enviados al módulo de streaming de audio. Éste reproducirá los datos por el altavoz del dispositivo.

A2.1.5. Mostrar la dirección IP del dispositivo

Se ha incluido una función para obtener la dirección IP local. Esta función es útil a la hora de crear el enlace entre dos participantes, ya que al utilizar conexiones GPRS, no se permite asignar direcciones estáticas.

La aplicación muestra la dirección IP del dispositivo mediante un cuadro de diálogo.

A2.2. Arquitectura

A continuación se muestran los distintos componentes que forman la aplicación.

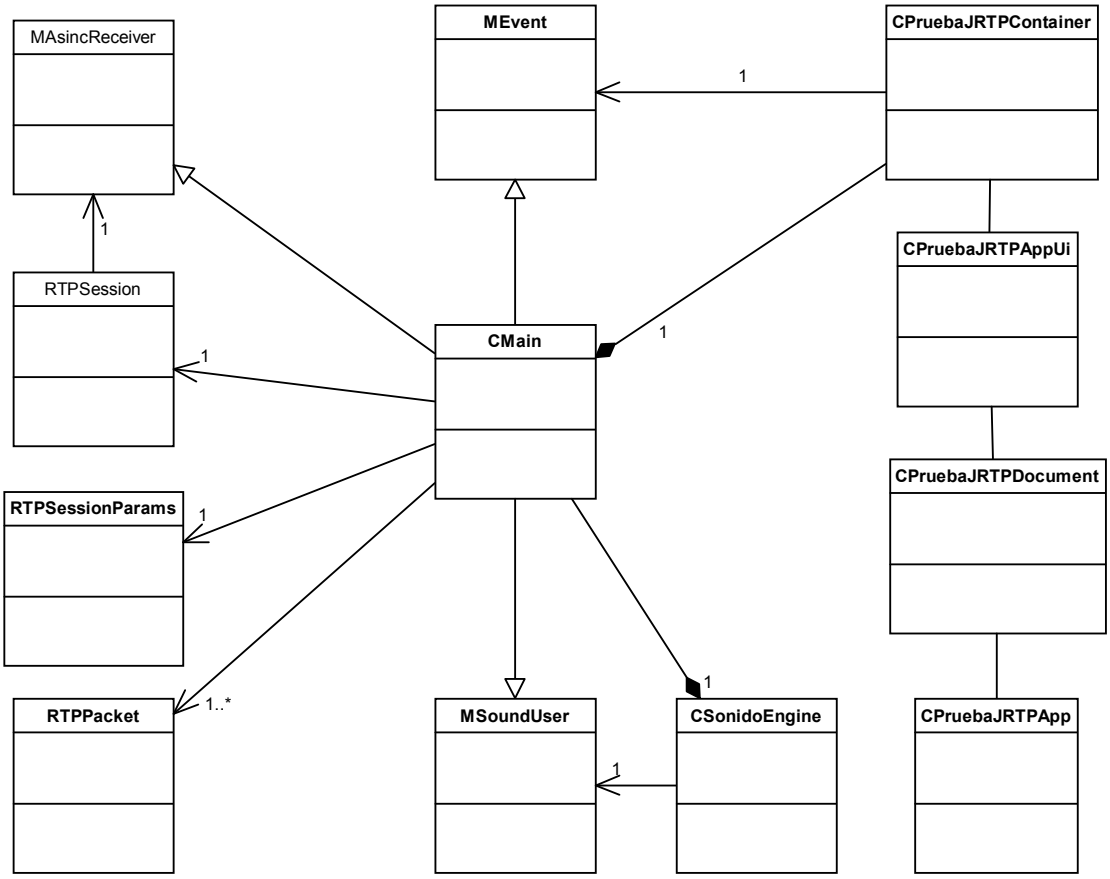

Fig. A2.4. Diagrama de clases de la aplicación.

La aplicación está formada por cuatro partes muy diferenciadas. Estas son: el *engine* de la aplicación, representado por la clase `CMain`, la librería RTP, el módulo de streaming de audio y la interfaz gráfica.

A2.2.1. Engine de la aplicación

Todas las operaciones de la aplicación se ejecutan en la clase `CMain`. Utilizando la nomenclatura de Symbian, esta clase se denomina *engine* de la aplicación. Contiene un conjunto de métodos, que permiten llevar a cabo todas las funcionalidades mostradas en el apartado A3.1. Para ello debe comunicarse con todos los elementos que forman la aplicación.

A2.2.1.1. Interacción con la librería RTP

Esta clase contiene una instancia a la clase `RTPSession`, la cual nos permite acceder a todas las funcionalidades de la librería RTP. Además contiene una instancia a `RTPSessionParams`, utilizada para establecer los parámetros de la sesión RTP. Asimismo, se crean diferentes instancias de la clase `RTPPacket` para poder recibir los datos entrantes.

Para poder recibir notificaciones de la llegada de nuevos datos entrantes, esta clase debe heredar de la clase abstracta `MasincReceiver`. Esta interfaz contiene un método virtual, que tiene que ser implementado en la aplicación, de forma que la librería pueda llamarlo cada vez que hayan nuevos datos. Esta aplicación se ha implementado de forma que, al ser llamado este método se recorren las fuentes y se obtienen los datos, tratándolos de la forma adecuada, según se haya configurado la recepción de la información (cadenas de texto o audio).

A2.2.1.2. Interacción con el módulo de streaming de audio

El módulo de audio está representado por la clase `CsonidoEngine`, por lo tanto la aplicación contiene una instancia a esta clase.

Cuando la aplicación desea reproducir datos por el altavoz del dispositivo, únicamente tiene que llamar al método `Play()` de la instancia al módulo de audio. De esta forma, cuando se ha configurado la aplicación para que reciba audio, el payload de los paquetes recibidos se pasa a este módulo, generando un stream de audio en la salida del dispositivo.

Para enviar paquetes de audio, el *engine* de la aplicación debe indicar al módulo de audio que empiece a capturar. A la hora de recibir el stream que genera el módulo de audio, se ha seguido el modelo de captura de eventos de Symbian OS. La clase `CMain` debe heredar de la interfaz `MsoundUser`, implementando un método virtual. De esta forma, cuando el módulo de streaming captura un fragmento de audio, llama a este método, pasando dicho fragmento. En esta aplicación, este método se ha implementado de forma que, cuando es llamado por el módulo de audio, se envía el fragmento de voz a través de la sesión RTP.

A2.2.1.3. Interacción con la interfaz gráfica

Las clases que forman la interfaz gráfica contienen una instancia de la clase `CMain`. De esta forma, cuando el usuario genera una petición, estas clases se encargan de llamar al método apropiado del *engine* de al aplicación.

La comunicación inversa, es decir, del *engine* hacía la interfaz gráfica se realiza a través de eventos. La clase `CPruebaJRTPContainer` debe heredar de la clase abstracta `MEvent`. Esta clase implementa los tres métodos virtuales que define la interfaz, los cuales permiten que la interfaz gráfica reciba cadenas de texto para ser mostradas por pantalla. `CMain` contiene una referencia a esta clase abstracta, pudiendo así enviar estas notificaciones a la aplicación. Dependiendo de que tipo de notificación se quiera mostrar, se utiliza uno de los tras métodos: el primero para notificaciones de estado, otro para notificaciones de recepción y un tercero para notificaciones de envío. La interfaz gráfica muestra el texto en la etiqueta correspondiente.

A2.2.2. Librería RTP

Para dotar a la aplicación de conectividad RTP, se ha incluido la librería RTP implementada en el proyecto. La principal representación de ésta es la clase `RTPSession`, por lo que la aplicación debe contener una instancia a esta clase. Adicionalmente, la aplicación debe utilizar las clases `RTPSessionParams` y `RTPPacket`, también definidas en la librería RTP, así como heredar de la interfaz `MasincReceiver`. A continuación se muestra el procedimiento que la aplicación debe seguir para utilizar la librería:

- Captura de eventos: Al heredar de `MasincReceiver` se debe implementar el método virtual `DataReceived`(), que será utilizado por la librería para generar el evento que informa de la llegada de datos entrantes. El contenido de este método es libre para el desarrollador, ya que no es obligatorio obtener los datos en la ejecución de este método.

- Configuración de la sesión: Como paso previo a la creación de la sesión RTP, se deben establecer los parámetros de ésta. Para ello se debe utilizar un método de la clase `RTPSessionParams`: `SetOwnTimestampUnit`(). Este método configura una unidad propia de timestamp. Adicionalmente, se debe utilizar el método `SetPortbase`() de la clase `RTPUDPv4TransmissionParams` para indicar el puerto local que permanecerá a la espera de datos.

- Creación de la sesión RTP: Una vez configurado el timestamp, se procede a la creación de la sesión RTP. Es necesario llamar al método `Create`() de la clase `RTPSession`. Esta función recibe como parámetros las instancias de las clases implicadas en la configuración. Para finalizar, se debe llamar al método `AddAsincUser`, pasando como parámetro una instancia de la propia aplicación. De esta forma la sesión RTP podrá llamar al método virtual `DataReceived`(), generando así un evento que podrá ser capturado por la aplicación.

- Adición de destinatarios: Si se desean añadir destinatario a la lista es necesario utilizar el método `AddDestination`(), pasando como parámetro un objeto del tipo `RTPIPv4Address`, el cual deberá contener la dirección IP y el puerto del destinatario.

- Empezar a recibir: Para que la librería empiece a escuchar por el puerto local, a la espera de nuevos datos entrantes, se debe realizar una llamada a `StartReading`().

- Envío de datos: Para enviar datos en paquetes RTP, únicamente hay que hacer una llamada al método `SendPacket`(), pasándole los datos a enviar.

- Consulta de datos recibidos: La librería RTP ofrece un mecanismo de acceso a los datos entrantes basado en listas. Para poder obtener los paquetes recibidos es necesario seguir los siguientes pasos: Al iniciar y terminar la consulta se debe llamar a los métodos `BeginDataAcces`() y `EndDataAccess`() respectivamente. Esto ofrece un mecanismo de protección de la información. A continuación es necesario llamar al método `GotoFirstSourceWithData`(). Si éste retorna *true* significa que hay alguna fuente con nuevos datos entrantes. En este caso la aplicación debe crear una instancia a la clase `RTPPacket` para poder parsear estos datos. A continuación se crea una estructura iterativa, que permite obtener todos los paquetes entrantes de una fuente, utilizando el método `GetNextPacket`(). Esto se ha de repetir para todas las fuentes con datos, pudiendo recorrerlas gracias al método `GotoNextSourceWithData`(). A continuación se muestra un diagrama de flujo, en el que se ilustra un posible algoritmo para el acceso a los datos.

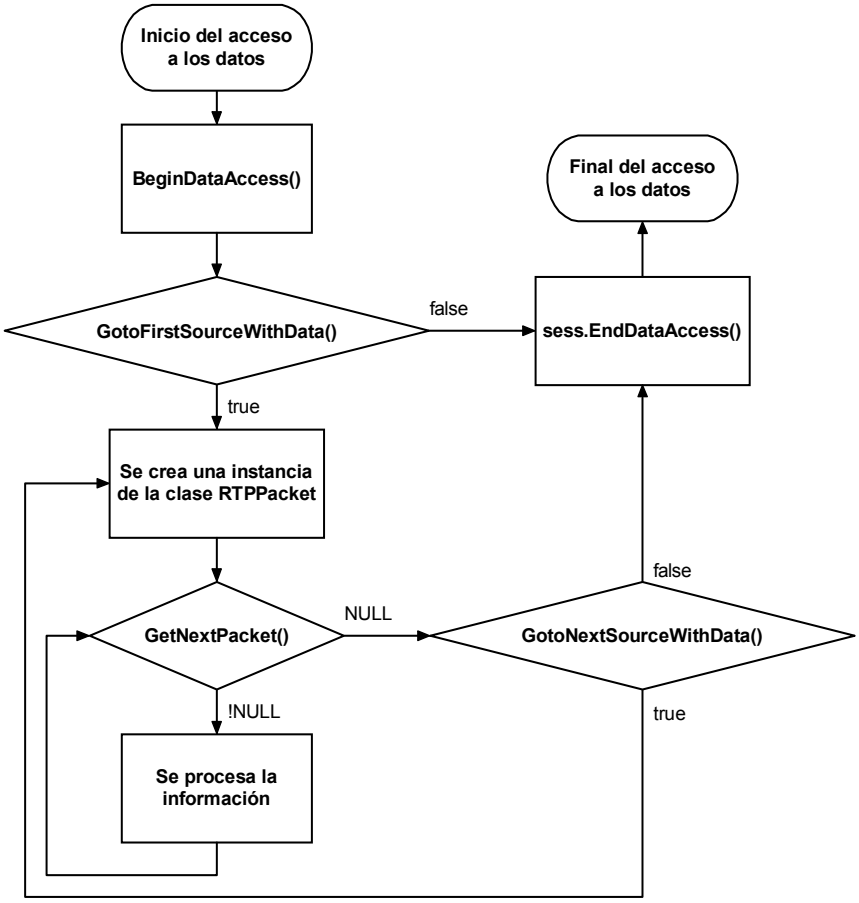

Fig. A2.5. Diagrama de flujo del acceso a datos.

A2.2.3. Módulo de streaming de audio

Con el fin de dotar a la aplicación de funciones multimedia, se ha añadido el módulo de streaming de audio. La utilización de éste es muy sencilla. La

aplicación deberá heredar de la clase abstracta `MsoundUser` e implementar la función virtual `SoundReceived`(). De esta forma se permite al *engine* de audio la posibilidad de entregar el stream generado a la aplicación, a través de eventos. Para que se empiece a generar este stream hay que llamar a la función `StartReading`().

Para enviar datos al *engine* y que éste los reproduzca por el altavoz del dispositivo, únicamente es necesario llamar al método `Play`(), pasándole los datos.

A2.2.4. Interfaz gráfica

La interfaz gráfica de las aplicaciones en Symbian OS está formada por cuatro clases. La motivación de su uso es proporcionar compatibilidad entre las distintas plataformas Symbian. Al separar las funciones de la interfaz, la adaptación de ésta a una nueva plataforma se puede llevar a cabo dejando intacta la implementación de algunas de estas clases. Esto facilita la portabilidad de las aplicaciones.

Para esta aplicación se ha creado una sencilla interfaz gráfica, que permite la interacción con el usuario a través de un sistema de menús y submenús. La descripción de las clases utilizadas es la siguiente:

- Clase CPruebaJRTPApp: Ésta también es conocida como clase aplicación. Posee dos funciones principales: definir las propiedades de la aplicación y crear una nueva clase documento. En esta clase se define el UID, identificador único de la aplicación.

- Clase CPruebaJRTPDocument: También conocida como clase documento, representa el modelo de datos para la aplicación. Si la aplicación está basada en ficheros, el documento es responsable de guardar y restaurar los datos de la aplicación.

- Clase CPruebaJRTPAppui: Esta clase se encarga de crear y gestionar las vistas presentes en la aplicación, soportando el dibujado y la interacción basada en ventanas. Es en esta clase dónde se procesan los eventos generados por el usuario.

- Clase CPruebaJRTPContainer: Esta clase posee los métodos encargados para el dibujado. Asimismo, esta clase hace de punto de entrada para el *engine* de la aplicación hacía la interfaz gráfica.